Everything curl://

The complete guide to all there is to know about the curl project

Daniel Stenberg

Table of Contents

Introduction

Everything curl is an extensive guide to everything there is to know about curl, the project, the command-line tool, the library, how everything started and how it came to be what it is today. How we work on developing it further, what it takes to use it, how you can contribute with code and bug reports and why all those millions of existing users use it.

This book is meant to be interesting and useful to both casual readers and the somewhat more experienced developers, and offers something for you all to pick and choose from. Don't read it from front to back. Read the chapters you are curious about and go back and forth as you see fit.

I hope to run this book project as I do all other projects I work on: in the open, completely free to download and read, free for anyone to comment on, available for everyone to contribute to and help out with. Send your bug reports, pull requests or critiques to me and I will improve this book accordingly.

This book will never be finished. I intend to keep working on it and while I may at some point in time consider it fairly complete and covering most aspects of the project (even if only that seems like an insurmountable goal), the curl project will continue to move so there will always be things to update in the book as well.

This book project started at the end of September 2015.

The book sites

https://bookcurl.haxx.se is the home of this book. It features easy accessible links to read the book online in a web version or download a copy for offline reading using one of the many different versions offered, including PDF, ePUB and MOBI.

https://ec.haxx.se is a shortcut to the HTML version of the book.

https://github.com/bagder/everything-curl hosts all the book content.

The author

With the hope of becoming just a co-author of this material, I am Daniel Stenberg. I founded the curl project. I'm a developer at heart, for fun and profit. I live and work in Stockholm, Sweden.

All there is to know about me can be found on my web site.

Help!

If you find mistakes, omissions, errors or blatant lies in this document, please send me a refreshed version of the affected paragraph and I will make amended versions. I will give proper credits to everyone who helps out! I hope to make this document better over time.

Preferably, you submit errors or pull requests on the book's github page.

License

How to read this book

Here is an overview of the main sections of this book and what they cover.

1. The cURL project

Project things. How it started, how we work and how often releases are made.

2. Open Source

An attempt to explain what open source is and how it works.

3. The source code

A description of the curl source tree and how the layout of the code is and works.

4. Network and protocols

What exactly are networks and protocols?

5. Command line basics

Start at the beginning. How do you use curl from a command line?

6. Using curl

Going deeper, looking at things you do with curl the command line tool.

7. How to HTTP with curl

Digging deeper on HTTP specific actions to do with the curl command line tool.

8. Building and installing

Explaining how you can build curl and libcurl from source code.

9. libcurl basics

How libcurl works and how you use it when writing your own applications with it.

10. HTTP with libcurl

A closer look at doing HTTP specific things with libcurl.

11. Bindings

A casual overview of some of the most popular libcurl bindings and how similar they are to the libcurl C API.

12. libcurl internals

Under the hood it works like this…

13. Index

The index.

The cURL project

A funny detail about Open Source projects is that they are called "projects", as if they were somehow limited in time or ever can get done. The cURL "project" is a number of loosely-coupled individual volunteers working on writing software together with a common mission: to do reliable data transfers with Internet protocols. And giving away the code for free for anyone to use.

How it started

Back in 1996, Daniel Stenberg was writing an IRC bot in his spare time, an automated program that would offer services for the participants in a chatroom dedicated to the Amiga computer (#amiga on the IRC network EFnet). He came to think that it would be fun to get some updated currency rates and have his bot offer a service online for the chat room users to get current exchange rates, to ask the bot "please exchange 200 USD into SEK" or similar.

In order to have the provided exchange rates as accurate as possible, the bot would download the rates daily from a web site that was hosting them. A small tool to download data over HTTP was needed for this task. A quick look-around at the time had Daniel find a tiny tool named httpget (written by a Brazilian named Rafael Sagula). It did the job, almost, just needed a few little a tweaks here and there and soon Daniel had taken over maintenance of the few hundred lines of code it was.

HttpGet 1.0 was subsequently released on April 8th 1997 with brand new HTTP proxy support.

We soon found and fixed support for getting currencies over GOPHER. Once FTP download support was added, the name of the project was changed and urlget 2.0 was released in August 1997. The http-only days were already passed.

The project slowly grew bigger. When upload capabilities were added and the name once again was misleading, a second name change was made and on March 20, 1998 curl 4 was released. (The version numbering from the previous names was kept.)

We consider **March 20 1998** to be curl's birthday.

The name

Naming things is hard.

The tool was about uploading and downloading data specified with a URL. It would show the data (by default). The user would "see" the URL perhaps and "see" then spelled with the single letter 'c'. It was also a client-side program, a URL client. So 'c' for Client and URL: **cURL**.

Nothing more was needed so the name was selected and we never looked back again.

Later on, someone suggested that curl could actually be a clever "recursive acronym" (where the first letter in the acronym refers back to the same word): "Curl URL Request Library"

While that is awesome, it was actually not the original thought. We sort of wish we were that clever though…

There are and were other projects using the name curl in various ways, but we were not aware of them by the time our curl came to be.

Pronunciation

Most of us pronounce "curl" with an initial k sound, just like the English word curl. It rhymes with words like girl and earl. Merriam Webster has a short WAV file to help.

Confusions and mixups

Soon after curl was first created another "curl" appeared that makes a programming language. That curl still exists.

Several libcurl bindings for various programming languages use the term "curl" or "CURL" in part or completely to describe their bindings, so sometimes you will find users talking about curl but targeting neither the command-line tool nor the library that is made by this project.

As a verb

'to curl something' is sometimes used as a reference to use a non-browser tool to download a file or resource from a URL.

What does curl do?

cURL is a project and its primary purpose and focus is to make two products:

- curl, the command-line tool
- libcurl the transfer library with a C API

Both the tool and the library do Internet transfers for resources specified as URLs using Internet protocols.

Everything and anything that is related to Internet protocol transfers can be considered curl's business. Things that are not related to that should be avoided and be left for other projects and products.

It could be important to also consider that curl and libcurl try to avoid handling the actual data that is transferred. It has, for example, no knowledge about HTML or anything else of the content that is popular to transfer over HTTP, but it knows all about how to transfer such data over HTTP.

Both products are frequently used not only to drive thousands or millions of scripts and applications for an Internet connected world, but they are also widely used for server testing, protocol fiddling and trying out new things.

The library is used in every imaginable sort of embedded device where Internet transfers are needed: car infotainment, televisions, Blu-Ray players, set-top boxes, printers, routers, game systems, etc.

Command line tool

Running curl from the command line was natural and Daniel never considered anything else than that it would output data on stdout, to the terminal, by default. The "everything is a pipe" mantra of standard Unix philosophy was something Daniel believed in. curl is like 'cat' or one of the other Unix tools; it sends data to stdout to make it easy to chain together with other tools to do what you want. That's also why virtually all curl options that allow reading from a file or writing to a file, also have the ability to select doing it to stdout or from stdin.

Following that style of what Unix command-line tools worked, it was also never any question about that it should support multiple URLs on the command line.

The command-line tool is designed to work perfectly from scripts or other automatic means. It doesn't feature any other GUI or UI other than mere text in and text out.

The library

While the command-line tool came first, the network engine was ripped out and converted into a library during the year 2000 and the concepts we still have today were introduced with libcurl 7.1 in August 2000. Since then, the command line tool has been a thin layer of logic to make a tool around the library that does all the heavy lifting.

libcurl is designed and meant to be available for anyone who wants to add client-side file transfer capabilities to their software, on any platform, any architecture and for any purpose. libcurl is also extremely liberally licensed to avoid that becoming an obstacle.

libcurl is written in traditional and conservative C. Where other languages are preferred, people have created libcurl bindings for them.

Project communication

cURL is an Open Source project consisting of voluntary members from all over the world, living and working in a large number of the world's time zones. To make such a setup actually work, communication and openness is key. We keep all communication public and we use open communication channels. Most discussions are held on mailing lists, we use bug trackers where all issues are discussed and handled with full insight for everyone who cares to look.

It is important to realize that we are all jointly taking care of the project, we fix problems and we add features. Sometimes a regular contributor grows bored and fades away, sometimes a new eager contributor steps out from the shadows and starts helping out more. To keep this ship going forward as well as possible, it is important that we maintain open discussions and that's one of the reasons why we frown upon users who take discussions privately or try to e-mail individual team members about development issues, questions, debugging or whatever.

In this day and age, mailing lists may be considered sort of the old style of communication—no fancy web forums or similar. Using a mailing list is therefore becoming an art that isn't practised everywhere and may be a bit strange and unusual to you. But fear not. It is just about sending emails to an address that then sends that e-mail out to all the subscribers. Our mailing lists have at most a few thousand subscribers. If you are mailing for the first time, it might be good to read a few old mails first to get to learn the culture and what's considered good practice.

The mailing lists and the bug tracker have changed hosting providers a few times and there are reasons to suspect it might happen again in the future. It is just the kind of thing that happens to a project that lives for a long time.

A few users also hang out on IRC in the #curl channel on freenode.

Mailing list etiquette

Like many communities and subcultures, we have developed guidelines and rules of what we think is the right way to behave and how to communicate on the mailing lists. The curl mailing list etiquette follows the style of traditional Open Source projects.

Do not mail a single individual

Many people send one question directly to one person. One person gets many mails, and there is only one person who can give you a reply. The question may be something that other people also want to ask. These other people have no way to read the reply but to ask the one person the question. The one person consequently gets overloaded with mail.

If you really want to contact an individual and perhaps pay for his or her services, by all means go ahead, but if it's just another curl question, take it to a suitable list instead.

Reply or new mail

Please do not reply to an existing message as a shortcut to post a message to the lists.

Many mail programs and web archivers use information within mails to keep them together as "threads", as collections of posts that discuss a certain subject. If you don't intend to reply on the same or similar subject, don't just hit reply on an existing mail and change subject; create a new mail.

Reply to the list

When replying to a message from the list, make sure that you do "group reply" or "reply to all", and not just reply to the author of the single mail you reply to.

We are actively discouraging replying back to the single person by setting the Reply-To: field in outgoing mails back to the mailing list address, making it harder for people to mail the author only by mistake.

Use a sensible subject

Please use a subject of the mail that makes sense and that is related to the contents of your mail. It makes it a lot easier to find your mail afterwards and it makes it easier to track mail threads and topics.

Do not top-post

If you reply to a message, don't use top-posting. Top-posting is when you write the new text at the top of a mail and you insert the previous quoted mail conversation below. It forces users to read the mail in a backwards order to properly understand it.

This is why top posting is so bad:

```
A: Because it messes up the order in which people normally read text.
Q: Why is top-posting such a bad thing?
A: Top-posting.
Q: What is the most annoying thing in e-mail?
```

Apart from the screwed-up read order (especially when mixed together in a thread when someone responds using the mandated bottom-posting style), it also makes it impossible to quote only parts of the original mail.

When you reply to a mail you let the mail client insert the previous mail quoted. Then you put the cursor on the first line of the mail and you move down through the mail, deleting all parts of the quotes that don't add context for your comments. When you want to add a comment you do so, inline, right after the quotes that relate to your comment. Then you continue downwards again.

When most of the quotes have been removed and you have added your own words, you are done!

HTML is not for mails

Please switch off those HTML encoded messages. You can mail all those funny mails to your friends. We speak plain text mails.

Quoting

Quote as little as possible. Just enough to provide the context you cannot leave out. A lengthy description can be found here.

Digest

We allow subscribers to subscribe to the "digest" version of the mailing lists. A digest is a collection of mails lumped together in one single mail.

Should you decide to reply to a mail sent out as a digest, there are two things you MUST consider if you really really cannot subscribe normally instead:

Cut off all mails and chatter that is not related to the mail you want to reply to.

Change the subject name to something sensible and related to the subject, preferably even the actual subject of the single mail you wanted to reply to.

Please tell us how you solved the problem!

Many people mail questions to the list, people spend some of their time and make an effort in providing good answers to these questions.

If you are the one who asks, please consider responding once more in case one of the hints was what solved your problems. The guys who write answers feel good to know that they provided a good answer and that you fixed the problem. Far too often, the person who asked the question is never heard of again, and we never get to know if he/she is gone because the problem was solved or perhaps because the problem was unsolvable!

Getting the solution posted also helps other users that experience the same problem(s). They get to see (possibly in the web archives) that the suggested fixes actually has helped at least one person.

Mailing lists

Some of the most important mailing lists are…

curl-users

The main mailing list for users and developers of the curl command-line tool, for questions and help around curl concepts, command-line options, the protocols curl can speak or even related tools. We tend to move development issues or more advanced bug fixes discussions over to curl-library instead, since libcurl is the engine that drives most of curl.

See https://cool.haxx.se/mailman/listinfo/curl-users

curl-library

The main development list, and also for users of libcurl. We discuss how to use libcurl in applications as well as development of libcurl itself. You will find lots of questions on libcurl behavior, debugging and documentation issues.

See https://cool.haxx.se/mailman/listinfo/curl-library

curl-announce

This mailing list only gets announcements about new releases and security problems—nothing else. This one is for those who want a more casual feed of information from the project. https://cool.haxx.se/mailman/listinfo/curl-announce

Reporting bugs

The development team does a lot of testing. We have a whole test suite that is run frequently every day on numerous platforms to in order to exercise all code and make sure everything works as supposed.

Still, there are times when things aren't working the way they should. Then we appreciate getting those problems reported.

A bug is a problem

Any problem can be considered a bug. A weirdly phrased wording in the manual that prevents you from understanding something is a bug. A surprising side effect of combining multiple options can be a bug—or perhaps it should be better documented? Perhaps the option doesn't do at all what you expected it to? That's a problem and we should fix it!

Problems must be known to get fixed

This may sound easy and uncomplicated but is a fundamental truth in our and other projects. Just because it is an old project and have thousands of users doesn't mean that the development team knows about the problem you just fell over. Maybe users haven't paid enough attention to details like you, or perhaps it just never triggered for anyone else.

We rely on users experiencing problems to report them. We need to learn the problems exist so that we can fix them.

Fixing the problems

Software engineering is, to a very large degree, about fixing problems. To fix a problem a developer needs to understand how to repeat it and to do that the debugging person needs to be told what set of circumstances that made the problem trigger.

A good bug report

A good report explains what happened and what you thought was going to happen. Tell us exactly what versions of the different components you used and take us step by step through what you do to get the problem.

After you submit a bug report, you can expect there to be follow-up questions or perhaps requests that you try out varies things and tasks in order for the developer to be able to narrow down the suspects and make sure your problem is being cornered in properly.

A bug report that is submitted but is abandoned by the submitter risks getting closed if the developer fails to understand it, fails to reproduce it or faces other problems when working on it. Don't abandon your report!

Report curl bugs in the curl bug tracker on github!

Testing

Testing software thoroughly and properly is a lot of work. Testing software that runs on dozens on operating systems and dozens of CPU architectures, with server implementations with their owns sets of bugs and interpretations of the specs, is even more work.

The curl project has a test suite that iterates over all existing test cases, runs the test and verifies that the outcome is the correct one and that no other problem happened, like a memory leak or something fishy in the protocol layer.

The test suite is meant to be possible to run after you have built curl yourself and there are a fair number of volunteers who also help out by running the test suite automatically a few times per day to make sure the latest commits get a run. This way, we hopefully discover the worst flaws pretty soon after they were introduced.

We don't test everything and even when we try to test things there will always be subtle details that get through and that we, sometimes years after the fact, figure out were wrong.

Due to the nature of different systems and funny use cases on the Internet, eventually some of the best testing is done by users when they run the code to perform their own use cases.

Another limiting factor with the test suite is that the test setup itself is less portable than curl and libcurl so there are in fact platforms where curl runs fine but the test suite cannot execute at all.

Releases

A release in the curl project means packaging up all the source code that is in the master branch of the code repository, signing the package, tagging the point in time in the code repository, and then putting it up on the web site for the world to download.

It is one single source code archive for all platforms curl can run on. It is the one and only package for both curl and libcurl.

We never ship any curl or libcurl *binaries* from the project. All the packaged binaries that are provided with operating systems or on other download sites are done by gracious volunteers outside of the project.

As of a few years back, we make an effort to do our releases on an eight week cycle and unless some really serious and urgent problem shows up we stick to this schedule. We release on a Wednesday, and then again a Wednesday eight weeks later and so it continues. Non-stop.

For every release we tag the source code in the repository with "curl-release version" and we update the changelog.

We had done 172 curl releases by January 2018, and for all the ones made since late 1999 there are lots of release stats available in our curl release log.

Daily snapshots

Every single change to the source code is committed and pushed to the source code repository. This repository is hosted on github.com and is using git these days (but hasn't always been this way). When building curl off the repository, there are a few things you need to generate and setup that sometimes cause people some problems or just friction. To help with that, we provide daily snapshots.

The daily snapshots are generated daily (clever naming, right?) as if a release had been made at that point in time. It produces a package of all sources code and all files that are normally part of a release and puts it in a package and uploads it to a special place (https://curl.haxx.se/snapshots/) to allow interested people to get the very latest code to test, to experiment or whatever.

The snapshots are only kept for around 20 days until deleted.

Security

Security is a primary concern for us in the curl project. We take it seriously and we work hard on providing secure and safe implementations of all protocols and related code. As soon as we get knowledge about a security related problem or just a suspected problem, we deal with it and we will attempt to provide a fix and security notice no later than in the next pending release.

We use a responsible disclosure policy, meaning that we prefer to discuss and work on security fixes out of the public eye and we alert the vendors on the openwall.org list a few days before we announce the problem and fix to the world. This, in an attempt to shorten the time span the bad guys can take advantage of a problem until a fixed version has been deployed.

Past security problems

During the years we have had our fair share of security related problems. We work hard on documenting every problem thoroughly with all details listed and clearly stated to aid users. Users of curl should be able to figure out what problems their particular curl versions and use cases are vulnerable to.

To help with this, we present this waterfall chart showing how all vulnerabilities affect which curl versions and we have this complete list of all known security problems since the birth of this project.

Trust

For a software to conquer the world, it needs to be trusted. It takes trust to build more trust and it can all be broken down really fast if the foundation is proven to have cracks.

In the curl project we build trust for our users in a few different ways:

1. We are completely transparent about everything. Every decision, every discussion as well as every line of code are always public and done in the open.
2. We try hard to write reliable code. We write test cases, we review code, we document best practices and we have a style guide that helps us keep code consistent.
3. We stick to promises and guarantees as much as possible. We don't break APIs and we don't abandon support for old systems.
4. Security is of utmost importance and we take every reported incident very seriously and realize that we **must** fix all known problems and we need to do it responsibly. We do our best to not endanger our users.
5. We act like adults. We can be silly and we can joke around, but we do it responsibly and we follow our Code of Conduct. Everyone should be able to even trust us to behave.

The development team

Daniel Stenberg is the founder and self-proclaimed leader of the project. Everybody else that participates or contributes in the project has thus arrived at a later point in time. Some contributors worked for a while and then left again. Most contributors hang around only for a short while to get their bug fixed or feature merged or similar. Counting all contributors we know the names of, we have received help from more than 1700 individuals.

The list of people that have repeatedly shown up in discussions and commits during the last several years include these stellar individuals:

- Daniel Stenberg
- Steve Holme
- Jay Satiro
- Dan Fandrich
- Marc Hörsken
- Kamil Dudka
- Alessandro Ghedini
- Yang Tse
- Günter Knauf
- Tatsuhiro Tsujikawa
- Patrick Monnerat
- Nick Zitzmann
- Marcel Raad
- Michael Kaufmann

Users of curl



We used to say that there are a billion users of curl. It makes a good line to say but in reality we, of course, don't have any numbers that exact. We just estimate and guess based on observations and trends. It also depends on exactly what you would consider "a user" to be. Let's elaborate.

Open Source

The project being Open Source and very liberally licensed means that just about anyone can redistribute curl in source format or built into binary form.

Counting downloads

The curl command-line tool and the libcurl library are available for download for most operating systems via the curl web site, they are provided via third party installers to a bunch and and they come installed by default with yet more operating systems. This makes

counting downloads from the curl web site completely inappropriate as a means of measurement.

Finding users

So, we can't count downloads and anyone may redistribute it and nobody is forced to tell us they use curl. How can we figure out the numbers? How can we figure out the users? The answer is that we really can't with any decent level of accuracy.

Instead we rely on witness reports, circumstantial evidence, on findings on the Internet, the occasional "about box" or license agreement mentioning curl or that authors ask for help and tell us about their use.

The curl license says users need to repeat it somewhere, like in the documentation, but that's not easy for us to find in many cases and it's also not easy for us to do anything about should they decide not to follow the very small license requirement.

Command-line tool users

The command-line tool curl is widely used by programmers around the world in shell and batch scripts, to debug servers and to test out things. There's no doubt it is used by millions every day.

Embedded library

libcurl is what makes our project reach the really large volume of users. The ability to quickly and easily get client side file transfer abilities into your application is desirable for a lot of users, and then libcurl's great portability also helps: you can write more or less the same application on a wide variety of platforms and you can still keep using libcurl for transfers.

libcurl being written in C with no or just a few required dependencies also help to get it used in embedded systems.

libcurl is popularly used in smartphone operating systems, in car infotainment setups, in television sets, in set-top boxes, in audio and video equipment such as Blu-Ray players and higher-end receivers. It is often used in home routers and printers.

A fair number of best-selling games are also using libcurl, on Windows and game consoles.

In web site backends

The libcurl binding for PHP was one of, if not the, first bindings for libcurl to really catch on and get used widely. It quickly got adopted as a default way for PHP users to transfer data and as it has now been in that position for over a decade and PHP has turned out to be a fairly popular technology on the Internet (recent numbers indicated that something like a quarter of all sites on the Internet uses PHP).

A few really high-demand sites are using PHP and are using libcurl in the backend. Facebook and Yahoo are two such sites.

Famous users

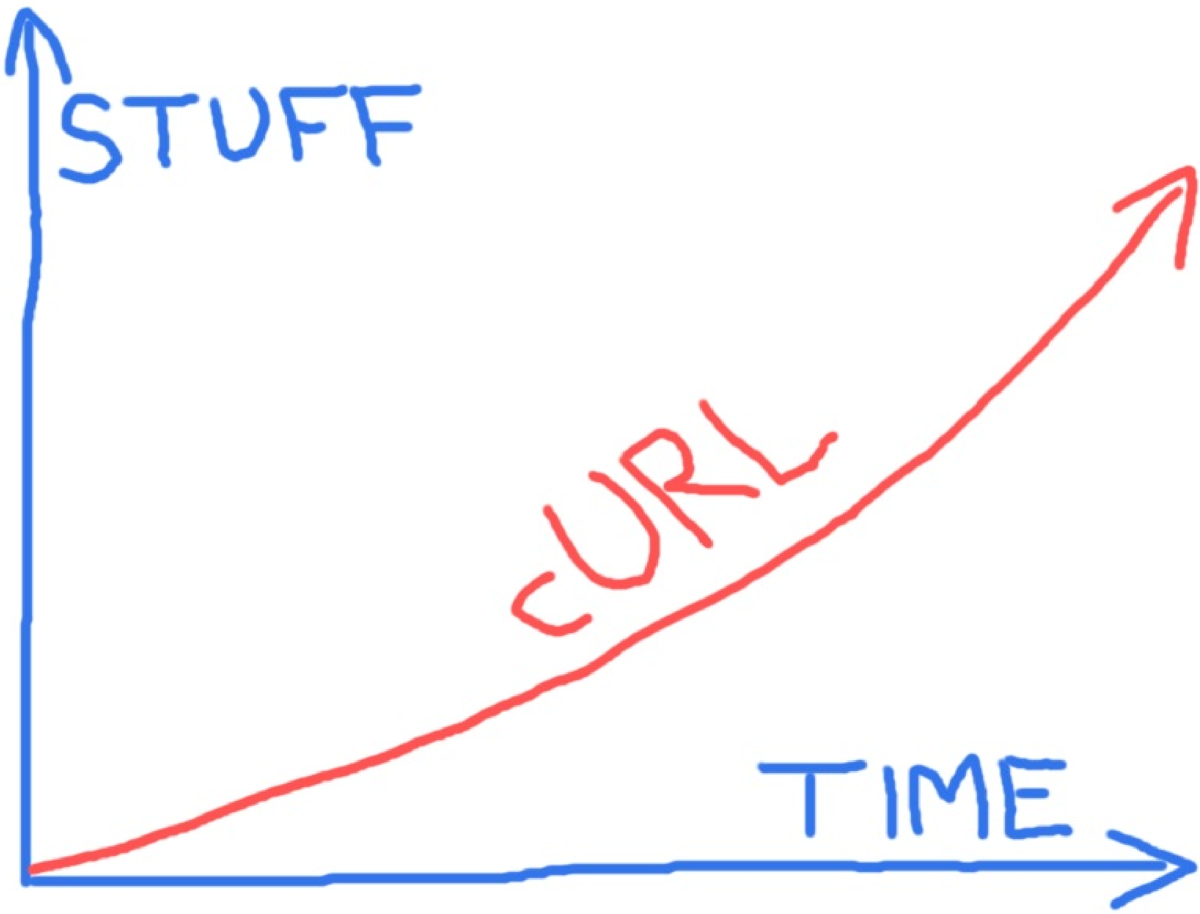
Nothing forces users to tell us they use curl or libcurl in their services or in the products. We usually only find out they do by accident, by reading about dialogues, documentation and license agreements. Of course some companies also just flat out tell us.

We collect names of companies and products on our web site of users that use the project's products "in commercial environments". We do this mostly just to show-off to other big brands that if these other guys can build products that depend on us, maybe you can, too?

The list of companies are well over 200 names, but extracting some of the larger or more well-known brands, here's a pretty good list that, of course, is only a small selection:

Adobe, Altera, AOL, Apple, AT&T, BBC, Blackberry, BMW, Bosch, Broadcom, Chevrolet, Cisco, Comcast, Facebook, Google, Hitachi, Honeywell, HP, Huawei, HTC, IBM, Intel, LG, Mazda, Mercedes-Benz, Motorola, Netflix, Nintendo, Oracle, Panasonic, Philips, Pioneer, RBS, Samsung, SanDisk, SAP, SAS Institute, SEB, Sharp, Siemens, Sony, Spotify, Sun, Swisscom, Tomtom, Toshiba, VMware, Xilinx, Yahoo, Yamaha

Future



There's no slowdown in sight in curl's future, bugs reported, development pace or how Internet protocols are being developed or updated.

We are looking forward to support for more protocols, support for more features within the already supported protocols, and more and better APIs for libcurl to allow users to do transfers even better and faster.

The project casually maintains a TODO file holding a bunch of ideas that we could work on in the future. It also keeps a KNOWN_BUGS document with, a list of known problems we would like to fix.

There's a ROADMAP document that describe some plans for the short-term that some of the active developers thought they would work on next. Of course, we can not promise that we will always follow it perfectly.

We are highly dependent on developers to join in and work on what they want to get done, be it bug fixes or new features.

Open Source

What is Open Source

Generally, Open Source software is software that can be freely accessed, used, changed, and shared (in modified or unmodified form) by anyone. Open Source software is typically made by many people, and distributed under licenses that comply with the definition.

Free Software is an older and related term that basically says the same thing for all our intents and purposes, but we stick to the term Open Source in this document for simplicity.

License

curl and libcurl are distributed under an Open Source license known as a MIT license derivative. It is very short, simple and easy to grasp. It follows here in full:

```
COPYRIGHT AND PERMISSION NOTICE

Copyright (c) 1996 - 2018, Daniel Stenberg, <daniel@haxx.se>.

All rights reserved.

Permission to use, copy, modify, and distribute this software for any purpose
with or without fee is hereby granted, provided that the above copyright
notice and this permission notice appear in all copies.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF THIRD PARTY RIGHTS. IN
NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM,
DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR
OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE
OR OTHER DEALINGS IN THE SOFTWARE.

Except as contained in this notice, the name of a copyright holder shall not
be used in advertising or otherwise to promote the sale, use or other dealings
in this Software without prior written authorization of the copyright holder.
```

This is basically legalese that says you are allowed to change the code, redistribute the code, redistribute binaries built from the code and build proprietary code with it, without anyone requiring you to give any changes back to the project—but you may not claim that you wrote it.

Early on in the project we iterated over a few different other licenses before we settled on this. We started out GPL, then tried MPL and landed on this MIT derivative. We will never change the license again.

Copyright

Copyright is a legal right granted by the law of a country that gives the creator of an original work exclusive rights for its use and distribution.

The copyright owner(s) can agree to allow others to use their work by licensing it. That's what we do in the curl project. The copyright is the foundation on which the licensing works.

Daniel Stenberg is the owner of most copyrights in the curl project.

Independent

A lot of Open Source projects are run within umbrella organizations. Such organizations include the GNU project, the Apache Software Foundation, a larger company that funds the project or similar. The curl project is not part of any such larger organization but is completely independent and free.

No company controls curl's destiny and the curl project does not need to follow any umbrella organization's guidelines.

curl is not a formal company, organization or a legal entity of any kind. curl is just an informal collection of humans, distributed across the globe, who work together on a software project.

Legal

The curl project obeys national laws of the countries in which it works. However, it is a highly visible international project, downloadable and usable in effectively every country on earth, so some local laws could be broken when using curl. That's just the nature of it and if uncertain, you should check your own local situation.

There have been law suits involving technology that curl provides. One such case known to the author of this was a patent case in the US that insisted they had the rights to resumed file transfers.

As a generic software component that is usable everywhere to everyone, there are times when libcurl—in particular—is used in nefarious or downright malicious ways. Examples include being used in virus and malware software. That is unfortunate but nothing we can prevent.

Code of conduct

As contributors and maintainers of this project, we pledge to respect all people who contribute through reporting issues, posting feature requests, updating documentation, submitting pull requests or patches, and other activities.

We are committed to making participation in this project a harassment-free experience for everyone, regardless of level of experience, gender, gender identity and expression, sexual orientation, disability, personal appearance, body size, race, ethnicity, age, or religion.

Examples of unacceptable behavior by participants include the use of sexual language or imagery, derogatory comments or personal attacks, trolling, public or private harassment, insults, or other unprofessional conduct.

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct. Project maintainers who do not follow the Code of Conduct may be removed from the project team.

This code of conduct applies both within project spaces and in public spaces when an individual is representing the project or its community.

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by opening an issue or contacting one or more of the project maintainers.

Development

We encourage everyone to participate in the development of curl and libcurl. We appreciate all the help we can get and while the main portion of this project is source code, there is a lot more than just coding and debugging help that is needed and useful.

We develop and discuss everything in the open, preferably on the mailing lists.

Source code on github

The source code to curl and libcurl have also been provided and published publicly and it continues to be uploaded to the main web site for every release.

Since March 2010, the curl source code repository has been hosted on github.com. By being up to date with the changes there, you can follow our day to day development very closely.

The source code

The source code is, of course, the actual engine parts of this project. After all, it is a software project.

curl and libcurl are written in C.

Hosting and download

You can always find the source code for the latest curl and libcurl release on the official curl web site. From there you can also find alternative mirrors that host copies and there are checksums and digital signatures provided to help you verify that what ends up on your local system when you download these files are the same bytes in the same order as were originally uploaded there by the curl team.

If you instead would rather work directly with the curl source code off our source code repository, you find all details in the curl github repository.

Clone the code

```
git clone https://github.com/curl/curl.git
```

This will get the latest curl code downloaded and unpacked in a directory on your local system.

Code layout

The curl source code tree is neither large nor complicated. A key thing to remember is, perhaps, that libcurl is the library and that library is the biggest component of the curl command-line tool.

root

We try to keep the number of files in the source tree root to a minimum. You will see a slight difference in files if you check a release archive compared to what is stored in the git repository as several files are generated by the release scripts.

Some of the more notable ones include:

- `buildconf` : used to build configure and more when building curl from source out of the git repository.
- `buildconf.bat` : the Windows version of buildconf. Run this after having checked out the full source code from git.
- `CHANGES` : generated at release and put into the release archive. It contains the 1000 latest changes to the source repository.
- `configure` : a generated script that is used on Unix-like systems to generate a setup when building curl.
- `COPYING` : the license detailing the rules for your using the code.
- `GIT-INFO` : only present in git and contains information about how to build curl after having checked out the code from git.
- `maketgz` : the script used to produce release archives and daily snapshots
- `README` : a short summary of what curl and libcurl are.
- `RELEASE-NOTES` : contains the changes done for the latest release; when found in git it contains the changes done since the previous release that are destined to end up in the coming release.

lib

This directory contains the full source code for libcurl. It is the same source code for all platforms—over one hundred C source files and a few more private header files. The header files used when building applications against libcurl are not stored in this directory; see include/curl for those.

Depending on what features are enabled in your own build and what functions your platform provides, some of the source files or portions of the source files may contain code that is not used in your particular build.

lib/vtls

The VTLS sub section within libcurl is the home of all the TLS backends libcurl can be built to support. The "virtual" TLS internal API is a common API that is used within libcurl to access TLS and crypto functions without the main code knowing exactly which TLS library that is used. This allows the person who builds libcurl to select from a wide variety TLS libraries to build with.

We also maintain a SSL comparison table on the web site to aid users.

- OpenSSL: the (by far) most popular TLS library.
- BoringSSL: an OpenSSL fork maintained by Google. It will make libcurl disable a few features due to lacking some functionality in the library.
- LibreSSL: an OpenSSL fork maintained by the OpenBSD team.
- NSS: a full-blown TLS library perhaps most known for being used by the Firefox web browser. This is the default TLS backend for curl on Fedora and Redhat systems.
- GnuTLS: a full-blown TLS library used by default by the Debian packaged curl.
- mbedTLS: (formerly known as PolarSSL) is a TLS library more targeted towards the embedded market.
- WolfSSL: (formerly known as cyaSSL) is a TLS library more targeted towards the embedded market.
- axTLS: a minuscule TLS library focused on a requiring a small footprint.
- Schannel: the native TLS library on Windows.
- SecureTransport: the native TLS library on Mac OS X.
- GSKit: the native TLS library on OS/400.

src

This directory holds the source code for the curl command-line tool. It is the same source code for all platforms that run the tool.

Most of what the command-line tool does is to convert given command line options into the corresponding libcurl options or set of options and then makes sure to issue them correctly to drive the network transfer according to the user's wishes.

This code uses libcurl just as any other application would.

include/curl

Here are the public header files that are provided for libcurl-using applications. Some of them are generated at configure or release time so they do not look identical in the git repository as they do in a release archive.

With modern libcurl, all an application is expected to include in its C source code is `#include <curl/curl.h>`

docs

The main documentation location. Text files in this directory are typically plain text files. We have slowly started to move towards Markdown format so a few (but hopefully growing number of) files use the .md extension to signify that.

Most of these documents are also shown on the curl web site automatically converted from text to a web friendly format/look.

- `BINDINGS` : lists all known libcurl language bindings and where to find them
- `BUGS` : how to report bugs and where
- `CODE_OF_CONDUCT.md` : how we expect people to behave in this project
- `CONTRIBUTE` : what to think about when contributing to the project
- `curl.1` : the curl command-line tool man page, in nroff format
- `curl-config.1` : the curl-config man page, in nroff format
- `FAQ` : frequently asked questions about various curl-related subjects
- `FEATURES` : an incomplete list of curl features
- `HISTORY` : describes how the project started and has evolved over the years
- `HTTP2.md` : how to use HTTP/2 with curl and libcurl
- `HTTP-COOKIES` : how curl supports and works with HTTP cookies
- `index.html` : a basic HTML page as a documentation index page
- `INSTALL` : how to build and install curl and libcurl from source
- `INSTALL.cmake` : how to build curl and libcurl with CMake
- `INSTALL.devcpp` : how to build curl and libcurl with devcpp
- `INTERNALS` : details curl and libcurl internal structures
- `KNOWN_BUGS` : list of known bugs and problems
- `LICENSE-MIXING` : describes how to combine different third party modules and their individual licenses
- `MAIL-ETIQUETTE` : this is how to communicate on our mailing lists
- `MANUAL` : a tutorial-like guide on how to use curl
- `mk-ca-bundle.1` : the mk-ca-bundle tool man page, in nroff format
- `README.cmake` : CMake-specific details
- `README.netware` : Netware-specific details
- `README.win32` : win32-specific details
- `RELEASE-PROCEDURE` : how to do a curl and libcurl release

- `RESOURCES` : further resources for further reading on what, why and how curl does things
- `ROADMAP.md` : what we want to work on in the future
- `SECURITY` : how we work on security vulnerabilities
- `SSLCERTS` : TLS certificate handling documented
- `SSL-PROBLEMS` : common SSL problems and their causes
- `THANKS` : thanks to this extensive list of friendly people, curl exists today!
- `TheArtOfHttpScripting` : a tutorial into HTTP scripting with curl
- `TODO` : things we or you can work on implementing
- `VERSIONS` : how the version numbering of libcurl works

docs/libcurl

All libcurl functions have their own man pages in individual files with .3 extensions, using nroff format, in this directory. The are also a few other files that are described below.

- `ABI`
- `index.html`
- `libcurl.3`
- `libcurl-easy.3`
- `libcurl-errors.3`
- `libcurl.m4`
- `libcurl-multi.3`
- `libcurl-share.3`
- `libcurl-thread.3`
- `libcurl-tutorial.3`
- `symbols-in-versions`

docs/libcurl/opts

This directory contains the man pages for the individual options for three different libcurl functions.

`curl_easy_setopt()` options start with `CURLOPT_` , `curl_multi_setopt()` options start with `CURLMOPT_` and `curl_easy_getinfo()` options start with `CURLINFO_` .

docs/examples

Contains around 100 stand-alone examples that are meant to help readers understand how libcurl can be used.

See also the libcurl examples section of this book.

scripts

Handy scripts.

- `contributors.sh` : extracts all contributors from the git repository since a given hash/tag. The purpose is to generate a list for the RELEASE-NOTES file and to allow manually added names to remain in there even on updates. The script uses the 'THANKS-filter` file to rewrite some names.
- `contrithanks.sh` : extracts contributors from the git repository since a given hash/tag, filters out all the names that are already mentioned in `THANKS` , and then outputs `THANKS` to stdout with the list of new contributors appended at the end; it's meant to allow easier updates of the THANKS document. The script uses the 'THANKS-filter` file to rewrite some names.
- `log2changes.pl` : generates the `CHANGES` file for releases, as used by the release script. It simply converts git log output.
- `zsh.pl` : helper script to provide curl command-line completions to users of the zsh shell.

Handling different build options

The curl and libcurl source code have been carefully written to build and run on virtually every computer platform in existence. This can only be done through hard work and by adhering to a few guidelines (and, of course, a fair amount of testing).

A golden rule is to always add #ifdefs that checks for specific features, and then have the setup scripts (configure or CMake or hard-coded) check for the presence of said features in a user's computer setup before the program is compiled there. Additionally and as a bonus, thanks to this way of writing the code, some features can be explicitly turned off even if they are present in the system and *could* be used. Examples of that would be when users want to, for example, build a version of the library with a smaller footprint or with support for certain protocols disabled, etc.

The project sometimes uses #ifdef protection around entire source files when, for example, a single file is provided for a specific operating system or perhaps for a specific feature that isn't always present. This is to make it possible for all platforms to always build all files—it simplifies the build scripts and makefiles a lot. A file entirely #ifdefed out hardly adds anything to the build time, anyway.

Rather than sprinkling the code with #ifdefs, to the extent where it is possible, we provide functions and macros that make the code look and work the same, independent of present features. Some of those are then empty macros for the builds that lack the features.

Both TLS handling and name resolving are handled with an internal API that hides the specific implementation and choice of 3rd party software library. That way, most of the internals work the same independent of which TLS library or name resolving system libcurl is told to use.

Style and code requirements

Source code that has a common style is easier to read than code that uses different styles in different places. It helps making the code feel like one continuous code base. Easy-to-read is a very important property of code and helps make it easier to review when new things are added and it helps debugging code when developers are trying to figure out why things go wrong. A unified style is more important than individual contributors having their own personal tastes satisfied.

Our C code has a few style rules. Most of them are verified and upheld by the lib/checksrc.pl script. Invoked with `make checksrc` or even by default by the build system when built after `./configure --enable-debug` has been used.

It is normally not a problem for anyone to follow the guidelines as you just need to copy the style already used in the source code, and there are no particularly unusual rules in our set of rules.

We also work hard on writing code that is warning-free on all the major platforms and in general on as many platforms as possible. Code that obviously will cause warnings will not be accepted as-is.

Some of the rules that you won't be allowed to break are:

Indentation

We use only spaces for indentation, never TABs. We use two spaces for each new open brace.

Comments

Since we write C89 code, // aren't allowed. They weren't introduced in the C standard until C99. We use only /* and */ comments:

```
/* this is a comment */
```

Long lines

Source code in curl may never be wider than 80 columns. There are two reasons for maintaining this even in the modern era of very large and high resolution screens:

1. Narrower columns are easier to read than very wide ones. There's a reason newspapers have used columns for decades or centuries.
2. Narrower columns allow developers to more easily view multiple pieces of code next to each other in different windows. I often have two or three source code windows next to each other on the same screen, as well as multiple terminal and debugging windows.

Open brace on the same line

In if/while/do/for expressions, we write the open brace on the same line as the keyword and we then set the closing brace on the same indentation level as the initial keyword. Like this:

```
if(age < 40) {
  /* clearly a youngster */
}
```

else on the following line

When adding an `else` clause to a conditional expression using braces, we add it on a new line after the closing brace. Like this:

```
if(age < 40) {
  /* clearly a youngster */
}
else {
  /* probably intelligent */
}
```

No space before parentheses

When writing expressions using if/while/do/for, there shall be no space between the keyword and the open parenthesis. Like this:

```
while(1) {
  /* loop forever */
}
```

Contributing

Contributing means helping out.

When you contribute anything to the project—code, documentation, bug fixes, suggestions or just good advice—we assume you do this with permission and you are not breaking any contracts or laws by providing that to us. If you don't have permission, don't contribute it to us.

Contributing to a project like curl could be many different things. While source code is the stuff that is needed to build the products, we are also depending on good documentation, testing (both test code and test infrastructure), web content, user support and more.

Send your changes or suggestions to the team and by working together we can fix problems, improve functionality, clarify documentation, add features or make anything else you help out with land in the proper place. We will make sure improved code and docs get merged into the source tree properly and other sorts of contributions are suitable received.

Send your contributions on a mailing list, file an issue or submit a pull request.

Suggestions

Ideas are easy, implementations are hard. Yes, we do appreciate good ideas and suggestions of what to do and how to do it, but the chances that the ideas actually turn into real features grow substantially if you also volunteer to participate in converting the idea into reality.

We already gather ideas in the `TODO` document and we are generally aware of the current trends in the popular networking protocols so there is usually no need to remind us about those.

What to add

The best approach to add anything to curl or libcurl is, of course, to first bring the idea and suggestion to the curl project team members and then discuss with them if the idea is feasible for inclusion and then how an implementation is best done—and done in the best possible way to get merged into the source code repository, assuming that is what you want.

The project generally approves of functions that improves the support for the current protocols, especially features that popular clients or browsers have but that curl still lacks.

Of course, you can also add contents to the project that isn't code, like documentation, graphics or web site contents, but the general rules apply equally to that.

If you are fixing a problem you have or a problem that others are reporting, we will be thrilled to receive your fix and merge it as soon as possible!

What not to add

There aren't any good rules to say what features you can't add or that we will never accept, but let me instead try to mention a few things you should avoid to get less friction and to be successful, faster:

- Do not write up a huge patch first and then send it to the list for discussion. Always start out by discussing on the list, and send your initial review requests early to get feedback on your design and approach. It saves you from wasting time going down a route that might need rewriting in the end anyway!
- When introducing things in the code, you need to follow the style and architecture that already exists. When you add code to the ordinary transfer code path, it must, for example, work asynchronously in a non-blocking manner. We will not accept new code that introduces blocking behaviors—we already have too many of those that we haven't managed to remove yet.
- Quick hacks or dirty solutions that have a high risk of not working on platforms you don't run or on architectures you don't know. We don't care if you are in a hurry or that it works for you. We do not accept high risk code or code that is hard to read or understand.
- Code that breaks the build. Sure, we accept that we sometimes have to add code to certain areas that makes the new functionality perhaps depend on a specific 3rd party library or a specific operating system and similar, but we can **never** do that at the expense of all other systems. We don't break the build, and we make sure all tests keep running successfully.

git

Our preferred source control tool is git.

While git is sometimes not the easiest tool to learn and master, all the basic steps a casual developer and contributor needs to know are very straight-forward and do not take much time or effort to learn.

This book will not help you learn git. All software developers in this day and age should learn git anyway.

The curl git tree can be browsed with a web browser on our github page at https://github.com/curl/curl.

To check out the curl source code from git, you can clone it like this:

```
git clone https://github.com/curl/curl.git
```

Pull request

A very popular and convenient way to make your own changes and contribute them back to the project is by doing a so-called pull request on github.

First, you create your own version of the source tree, called a fork, on the github web site. That way you get your own version of the curl git tree that you can clone to a local copy.

You edit your own local copy, commit the changes, push them to the git repository on github and then on the github web site you can select to create a pull request based on your changes done to your local repository clone of the original curl repository.

We recommend doing your work meant for a pull request in a dedicated separate branch and not in master, just to make it easier for you to update a pull request, like after review, for example, or if you realize it was a dead end and you decide to just throw it away.

Make a patch for the mailing list

Even if you opt to not make a pull request but prefer the old fashioned and trusted method of sending a patch to the curl-library mailing list, it is still a good to work in a local git branch and commit your changes there.

A branch makes it easy to edit and rebase when you need to change things and it makes it easy to keep syncing to the master branch when things are updated upstream.

Once your commits are fine enough to get sent to the mailing list, you just create patches with `git format-patch` and send them away. Even more fancy users go directly to `git send-email` and have git send the e-mail itself!

git commit style

When you commit a patch to git, you give it a commit message that describes the change you are committing. We have a certain style in the project that we ask you to use:

```
[area]: [short line describing the main effect]

[separate the above single line from the rest with an empty line]

[full description, no wider than 72 columns that describes as much as
possible as to why this change is made, and possibly what things
it fixes and everything else that is related]

[Bug: link to source of the report or more related discussion]
[Reported-by: John Doe—credit the reporter]
[whatever-else-by: credit all helpers, finders, doers]
```

Don't forget to use `git commit --author="Jane Doe <jane@example.com>"` if you commit someone else's work, and make sure that you have your own user and e-mail setup correctly in git before you commit!

The author and the *-by: lines are, of course, there to make sure we give the proper credit in the project. We do not want to take someone else's work without clearly attributing where it comes from. Giving correct credit is of utmost importance!

Who decides what goes in?

First, it might not be obvious to everyone but there is, of course, only a limited set of people that can actually merge commits into the actual official git repository. Let's call them the core team.

Everyone else can fork off their own curl repository to which they can commit and push changes and host them online and build their own curl versions from and so on, but in order to get changes into the *official* repository they need to be pushed by a trusted person.

The core team is a small set of curl developers who have been around for a several years and that have shown that they are skilled developers and that they fully comprehend the values and the style of development we do in this project. They are some of the people listed in the The development team section.

You can always bring a discussion to the mailing list and motivation why you think your changes should get accepted, or perhaps even object to other changes that are getting in and so forth. You can even suggest yourself or someone else to be given "push rights" and become one of the selected few in that team.

Daniel remains the project leader and while it is very rarely needed, he has the final say in debates that don't seem to sway in either direction or fail to reach some sort of consensus.

Reporting vulnerabilities

All known and public curl or libcurl related vulnerabilities are listed on the curl web site security page.

Security vulnerabilities should not be entered in the project's public bug tracker unless the necessary configuration is in place to limit access to the issue to only the reporter and the project's security team.

Vulnerability handling

The typical process for handling a new security vulnerability is as follows.

No information should be made public about a vulnerability until it is formally announced at the end of this process. That means, for example, that a bug tracker entry must NOT be created to track the issue since that will make the issue public and it should not be discussed on any of the project's public mailing lists. Also messages associated with any commits should not make any reference to the security nature of the commit if done prior to the public announcement.

- The person discovering the issue, the reporter, reports the vulnerability privately to `curl-security@haxx.se` . That's an e-mail alias that reaches a handful of selected and trusted people.
- Messages that do not relate to the reporting or managing of an undisclosed security vulnerability in curl or libcurl are ignored and no further action is required.
- A person in the security team sends an e-mail to the original reporter to acknowledge the report.
- The security team investigates the report and either rejects it or accepts it.
- If the report is rejected, the team writes to the reporter to explain why.
- If the report is accepted, the team writes to the reporter to let him/her know it is accepted and that they are working on a fix.
- The security team discusses the problem, works out a fix, considers the impact of the problem and suggests a release schedule. This discussion should involve the reporter as much as possible.

- The release of the information should be "as soon as possible" and is most often synced with an upcoming release that contains the fix. If the reporter, or anyone else, thinks the next planned release is too far away then a separate earlier release for security reasons should be considered.
- Write a security advisory draft about the problem that explains what the problem is, its impact, which versions it affects, any solutions or workarounds and when the fix was released, making sure to credit all contributors properly.
- Request a CVE number from distros@openwall when also informing and preparing them for the upcoming public security vulnerability announcement—attach the advisory draft for information. Note that 'distros' won't accept an embargo longer than 19 days.
- Update the "security advisory" with the CVE number.
- The security team commits the fix in a private branch. The commit message should ideally contain the CVE number. This fix is usually also distributed to the 'distros' mailing list to allow them to use the fix prior to the public announcement.
- At the day of the next release, the private branch is merged into the master branch and pushed. Once pushed, the information is accessible to the public and the actual release should follow suit immediately afterwards.
- The project team creates a release that includes the fix.
- The project team announces the release and the vulnerability to the world in the same manner we always announce releases—it gets sent to the curl-announce, curl-library and curl-users mailing lists.
- The security web page on the web site should get the new vulnerability mentioned.

curl-security@haxx.se

Who is on this list? There are a couple of criteria you must meet, and then we might ask you to join the list or you can ask to join it. It really isn't very formal. We basically only require that you have a long-term presence in the curl project and you have shown an understanding for the project and its way of working. You must have been around for a good while and you should have no plans on vanishing in the near future.

We do not make the list of participants public mostly because it tends to vary somewhat over time and a list somewhere will only risk getting outdated.

Web site source code

Most of the curl web site is also available in a public git repository, although separate from the source code repository since it generally isn't interesting to the same people and we can maintain a different list of people that have push rights, etc.

The web site git repository is available on github at this URL: https://github.com/curl/curl-www and you can clone a copy of the web code like this:

```
git clone https://github.com/curl/curl-www.git
```

Building the web

The web site is an old custom-made setup that mostly builds static HTML files from a set of source files. The sources files are preprocessed with what is basically a souped-up C preprocessor called fcpp and a set of perl scripts. The man pages get converted to HTML with roffit. Make sure fcpp, perl, roffit, make and curl are all in your $PATH.

Once you have cloned the git repository the first time, invoke `sh bootstrap.sh` once to get a symlink and some some initial local files setup, and then you can build the web site locally by invoking `make` in the source root tree.

Note that this doesn't make you a complete web site mirror, as some scripts and files are only available on the real actual site, but should give you enough to let you view most HTML pages locally.

Network and protocols

Before diving in and talking about how to use curl to get things done, let's take a look at what all this networking is and how it works, using simplifications and some minor shortcuts to give an easy overview.

The basics are in the networking simplified chapter that tries to just draw a simple picture of what networking is from a curl perspective, and the protocols section which explains what exactly a "protocol" is and how that works.

Networking simplified

Networking means communicating between two endpoints on the Internet. The Internet is just a bunch of interconnected machines (computers really), each using their own private addresses (called IP addresses). The addresses each machine have can be of different types and machines can even have temporary addresses. These computers are often called hosts.

The computer, tablet or phone you sit in front of is usually called "the client" and the machine out there somewhere that you want to exchange data with is called "the server". The main difference between the client and the server is in the roles they play here. There's nothing that prevents the roles from being reversed in a subsequent operation.

Which machine

When you want to initiate a transfer to one of the machines out there (a server), you usually don't know its IP addresses but instead you usually know its name. The name of the machine you will talk to is embedded in the URL that you work with when you use curl.

You might use a URL like "http://example.com/index.html", which means you will connect to and communicate with the host named example.com.

Host name resolving

Once we know the host name, we need to figure out which IP addresses that host has so that we can contact it.

Converting the name to an IP address is often called 'name resolving'. The name is "resolved" to a set of addresses. This is usually done by a "DNS server", DNS being like a big lookup table that can convert names to addresses—all the names on the Internet, really. Your computer normally already knows the address of a computer that runs the DNS server as that is part of setting up the network.

curl will therefore ask the DNS server: "Hello, please give me all the addresses for example.com", and the server responds with a list of them. Or in the case you spell the name wrong, it can answer back that the name doesn't exist.

Establish a connection

With a list of IP addresses for the host curl wants to contact, curl sends out a "connect request". The connection curl wants to establish is called TCP and it works sort of like connecting an invisible string between two computers. Once established, it can be used to send a stream of data in both directions.

As curl gets a list of addresses for the host, it will actually traverse that list of addresses when connecting and in case one fails it will try to connect to the next one until either one works or they all fail.

Connects to "port numbers"

When connecting with TCP to a remote server, a client selects which port number to do that on. A port number is just a dedicated place for a particular service, which allows that same server to listen to other services on other port numbers at the same time.

Most common protocols have default port numbers that clients and servers use. For example, when using the "http://example.com/index.html" URL, that URL specifies a scheme called "http" which tells the client that it should try TCP port number 80 on the server by default. The URL can optionally provide another, custom, port number but if nothing special is specified, it will use the default port for the scheme used in the URL.

TLS

After the TCP connection has been established, many transfers will require that both sides negotiate a better security level before continuing, and that is often TLS; Transport Layer Security. If that is used, the client and server will do a TLS handshake first and only continue further if that succeeds.

Transfer data

When the connecting "string" we call TCP is attached to the remote computer (and we have done the possible additional TLS handshake), there's an established connection between the two machines and that connection can then be used to exchange data. That communication is done using a "protocol", as discussed in the following chapter.

Protocol

The language used to ask for data to get sent—in either direction—is called **the protocol**. The protocol describes exactly how to ask the server for data, or to tell the server that there is data coming.

Protocols are typically defined by the IETF (Internet Engineering Task Force), which hosts RFC documents that describe exactly how each protocol works: how clients and servers are supposed to act and what to send and so on.

What protocols does curl support?

curl supports protocols that allow "data transfers" in either or both directions. We usually also restrict ourselves to protocols which have a "URI format" described in an RFC or at least is somewhat widely used, as curl works primarily with URLs (URIs really) as the input key that specifies the transfer.

The latest curl (as of this writing) supports these protocols:

DICT, FILE, FTP, FTPS, GOPHER, HTTP, HTTPS, IMAP, IMAPS, LDAP, LDAPS, POP3, POP3S, RTMP, RTSP, SCP, SFTP, SMB, SMBS, SMTP, SMTPS, TELNET, TFTP

To complicate matters further, the protocols often exist in different versions or flavors as well.

What other protocols are there?

The world is full of protocols, both old and new. Old protocols get abandoned and dropped and new ones get introduced. There's never a state of stability but the situation changes from day to day and year to year. You can rest assured that there will be new protocols added in the list above in the future and that there will be new versions of the protocols already listed.

There are, of course, already other protocols in existence that curl doesn't yet support. We are open to supporting more protocols that suit the general curl paradigms, we just need developers to write the necessary code adjustments for them.

How are protocols developed?

Both new versions of existing protocols and entirely new protocols are usually developed by persons or teams that feel that the existing ones are not good enough. Something about them makes them not suitable for a particular use case or perhaps some new idea has popped up that could be applied to improve things.

Of course, nothing prevents anyone from developing a protocol entirely on their own at their own pleasure in their own backyard, but the major protocols are usually brought to the IETF at a fairly early stage where they are then discussed, refined, debated and polished and then eventually, hopefully, turned into a published RFC document.

Software developers then read the RFC specifications and deploy their code in the world based on their interpretations of the words in those documents. It sometimes turn out that some of the specifications are subject to vastly different interpretations or sometimes the engineers are just lazy and ignore sound advice in the specs and deploy something that doesn't adhere. Writing software that interoperates with other implementations of the specifications can therefore end up being hard work.

How much do protocols change?

Like software, protocol specifications are frequently updated and new protocol versions are created.

Most protocols allow some level of extensibility which makes new extensions show up over time, extensions that make sense to support.

The interpretation of a protocol sometimes changes even if the spec remains the same.

The protocols mentioned in this chapter are all "Application Protocols", which means they are transferred over more lower level protocols, like TCP, UDP and TLS. They are also themselves protocols that change over time, get new features and get attacked so that new ways of handling security, etc., forces curl to adapt and change.

About adhering to standards and who's right

Generally, there are protocol specs that tell us how to send and receive data for specific protocols. The protocol specs we follow are RFCs put together and published by IETF.

Some protocols are not properly documented in a final RFC, like, for example, SFTP for which our implementation is based on an Internet-draft that isn't even the last available one.

Protocols are, however, spoken by two parties and like in any given conversation, there are then two sides of understanding something or interpreting the given instructions in a spec. Also, lots of network software is written without the authors paying very close attention to the spec so they end up taking some shortcuts, or perhaps they just interpreted the text differently. Sometimes even mistakes and bugs make software behave in ways that are not mandated by the spec and sometimes even downright forbidden in the specs.

In the curl project we use the published specs as rules on how to act until we learn anything else. If popular alternative implementations act differently than what we think the spec says and that alternative behavior is what works widely on the big Internet, then chances are we will change foot and instead decide to act like those others. If a server refuses to talk with us when we think we follow the spec but works fine when we bend the rules every so slightly, then we probably end up bending them exactly that way—if we can still work successfully with other implementations.

Ultimately, it is a personal decision and up for discussion in every case where we think a spec and the real world don't align.

In the worst cases we introduce options to let application developers and curl users have the final say on what curl should do. I say worst because it is often really tough to ask users to make these decisions as it usually involves very tricky details and weirdness going on and it is a lot to ask of users. We should always do our very best to avoid pushing such protocol decisions to users.

The protocols curl supports

curl supports about 22 protocols. We say "about" because it depends on how you count and what you consider to be distinctly different protocols.

DICT

DICT is a dictionary network protocol, it allows clients to ask dictionary servers about a meaning or explanation for words. See RFC 2229. Dict servers and clients use TCP port 2628.

FILE

FILE is not actually a "network" protocol. It is a URL scheme that allows you to tell curl to get a file from the local file system instead of getting it over the network from a remote server. See RFC 1738.

FTP

FTP stands for File Transfer Protocol and is an old (originates in the early 1970s) way to transfer files back and forth between a client and a server. See RFC 959. It has been extended greatly over the years. FTP servers and clients use TCP port 21 plus one more port, though the second one is usually dynamically established during communication.

See the external page FTP vs HTTP for how it differs to HTTP.

FTPS

FTPS stands for Secure File Transfer Protocol. It follows the tradition of appending an 'S' to the protocol name to signify that the protocol is done like normal FTP but with an added SSL/TLS security layer. See RFC 4217.

This protocol is very problematic to use through firewalls and other network equipment.

GOPHER

Designed for "distributing, searching, and retrieving documents over the Internet", Gopher is somewhat of the grand father to HTTP as HTTP has mostly taken over completely for the same use cases. See RFC 1436. Gopher servers and clients use TCP port 70.

HTTP

The Hypertext Transfer Protocol, HTTP, is the most widely used protocol for transferring data on the web and over the Internet. See RFC 7230 for HTTP/1.1 and RFC 7540 for HTTP/2, the successor. HTTP servers and clients use TCP port 80.

HTTPS

Secure HTTP is HTTP done over an SSL/TLS connection. See RFC 2818. HTTPS servers and clients use TCP port 443.

IMAP

The Internet Message Access Protocol, IMAP, is a protocol for accessing, controlling and "reading" email. See RFC 3501. IMAP servers and clients use TCP port 143.

IMAPS

Secure IMAP is IMAP done over an SSL/TLS connection. Such connections usually start out as a "normal" IMAP connection that is then upgraded to IMAPS using the `STARTTLS` command.

LDAP

The Lightweight Directory Access Protocol, LDAP, is a protocol for accessing and maintaining distributed directory information. Basically a database lookup. See RFC 4511. LDAP servers and clients use TCP port 389.

LDAPS

Secure LDAP is LDAP done over an SSL/TLS connection.

POP3

The Post Office Protocol version 3 (POP3) is a protocol for retrieving email from a server. See RFC 1939. POP3 servers and clients use TCP port 110.

POP3S

Secure POP3 is POP3 done over an SSL/TLS connection. Such connections usually start out as a "normal" POP3 connection that is then upgraded to POP3S using the `STARTTLS` command.

RTMP

The Real-Time Messaging Protocol (RTMP) is a protocol for streaming audio, video and data. RTMP servers and clients use TCP port 1935.

RTSP

The Real Time Streaming Protocol (RTSP) is a network control protocol to control streaming media servers. See RFC 2326. RTSP servers and clients use TCP and UDP port 554.

SCP

The Secure Copy (SCP) protocol is designed to copy files to and from a remote SSH server. SCP servers and clients use TCP port 22.

SFTP

The SSH File Transfer Protocol (SFTP) that provides file access, file transfer, and file management over a reliable data stream. SFTP servers and clients use TCP port 22.

SMB

The Server Message Block (SMB) protocol is also known as CIFS. It is a an application-layer network protocol mainly used for providing shared access to files, printers, and serial ports and miscellaneous communications between nodes on a network. SMB servers and

clients use TCP port 485.

SMTP

The Simple Mail Transfer Protocol (SMTP) is a protocol for email transmission. See RFC 821. SMTP servers and clients use TCP port 25.

SMTPS

Secure SMTP is SMTP done over an SSL/TLS connection. Such connections usually start out as a "normal" SMTP connection that is then upgraded to SMTPS using the `STARTTLS` command.

TELNET

TELNET is an application layer protocol used over networks to provide a bidirectional interactive text-oriented communication facility using a virtual terminal connection. See RFC 854. TELNET servers and clients use TCP port 23.

TFTP

The Trivial File Transfer Protocol (TFTP) is a protocol for doing simple file transfers over UDP to get a file from or put a file onto a remote host. TFTP servers and clients use UDP port 69.

Command line basics

curl started out as a command-line tool and it has been invoked from shell prompts and from within scripts by thousands of users over the years. curl has established itself as one of those trusty tools that is there for you to help you get your work done.

Binaries and different platforms

The command-line tool "curl" is a binary executable file. The curl project does not by itself distribute or provide binaries. Binary files are highly system specific and oftentimes also bound to specific system versions.

To get a curl for your platform and your system, you need to get a curl executable from somewhere. Many people build their own from the source code provided by the curl project, lots of people install it using a package tool for their operating system and yet another portion of users download binary install packages from sources they trust.

No matter how you do it, make sure you are getting your version from a trusted source and that you verify digital signatures or the authenticity of the packages in other ways.

Also, remember that curl is often built to use third-party libraries to perform and unless curl is built to use them statically you must also have those third-party libraries installed; the exact set of libraries will vary depending on the particular build you get.

Command lines, quotes and aliases

There are many different command lines, shells and prompts in which curl can be used. They all come with their own sets of limitations, rules and guidelines to follow. The curl tool is designed to work with any of them without causing troubles but there may be times when your specific command line system doesn't match what others use or what is otherwise documented.

One way that command-line systems differ, for example, is how you can put quotes around arguments such as to embed spaces or special symbols. In most Unix-like shells you use double quotes (") and single quotes (') depending if you want to allow variable expansions or not within the quoted string, but on Windows there's no support for the single quote version.

In some environments, like PowerShell on Windows, the authors of the command line system decided they know better and "help" the user to use another tool instead of curl when `curl` is typed, by providing an alias that takes precedence when a command line is executed. In order to use curl properly with PowerShell, you need to type in its full name including the extension: "curl.exe".

Different command-line environments will also have different maximum command line lengths and force the users to limit how large amount of data that can be put into a single line. curl adapts to this by offering a way to provide command-line options through a file—or from stdin—using the -K option.

Garbage in, garbage out

curl has very little will of its own. It tries to please you and your wishes to a very large extent. It also means that it will try to play with what you give it. If you misspell an option, it might do something unintended. If you pass in a slightly illegal URL, chances are curl will still deal with it and proceed. It means that you can pass in crazy data in some options and you can have curl pass on that crazy data in its transfer operation.

This is a design choice, as it allows you to really tweak how curl does its protocol communications and you can have curl massage your server implementations in the most creative ways.

Command line options

When telling curl to do something, you invoke curl with zero, one or several command-line options to accompany the URL or set of URLs you want the transfer to be about. curl supports over two hundred different options.

Short options

Command line options pass on information to curl about how you want it to behave. Like you can ask curl to switch on verbose mode with the -v option:

```
curl -v http://example.com
```

-v is here used as a "short option". You write those with the minus symbol and a single letter immediately following it. Many options are just switches that switches something on or changes something between two known states. They can be used with just that option name. You can then also combine several single-letter options after the minus. To ask for both verbose mode and that curl follows HTTP redirects:

```
curl -vL http://example.com
```

The command-line parser in curl always parses the entire line and you can put the options anywhere you like; they can also appear after the URL:

```
curl http://example.com -Lv
```

Long options

Single-letter options are convenient since they are quick to write and use, but as there are only a limited number of letters in the alphabet and there are many things to control, not all options are available like that. Long option names are therefore provided for those. Also, as a convenience and to allow scripts to become more readable, most short options have longer name aliases.

Long options are always written with *two* minuses (or *dashes*, whichever you prefer to call them) and then the name and you can only write one option name per double-minus. Asking for verbose mode using the long option format looks like:

```
curl --verbose http://example.com
```

and asking for HTTP redirects as well using the long format looks like:

```
curl --verbose --location http://example.com
```

Arguments to options

Not all options are just simple boolean flags that enable or disable features. For some of them you need to pass on data, like perhaps a user name or a path to a file. You do this by writing first the option and then the argument, separated with a space. Like, for example, if you want to send an arbitrary string of data in an HTTP POST to a server:

```
curl -d arbitrary http://example.com
```

and it works the same way even if you use the long form of the option:

```
curl --data arbitrary http://example.com
```

When you use the short options with arguments, you can, in fact, also write the data without the space separator:

```
curl -darbitrary http://example.com
```

Arguments with spaces

At times you want to pass on an argument to an option, and that argument contains one or more spaces. For example you want to set the user-agent field curl uses to be exactly `I am your father`, including those three spaces. Then you need to put quotes around the string when you pass it to curl on the command line. The exact quotes to use varies depending on your shell/command prompt, but generally it will work with double quotes in most places:

```
curl -A "I am your father" http://example.com
```

Failing to use quotes, like if you would write the command line like this:

```
curl -A I am your father http://example.com
```

… will make curl only use 'I' as a user-agent string, and the following strings, 'am', your, etc will instead all be treated as separate URLs since they don't start with `-` to indicate that they're options and curl only ever handles options and URLs.

To make the string itself contain double quotes, which is common when you for example want to send a string of JSON to the server, you may need to use single quotes (except on Windows, where single quotes doesn't work the same way). Send the JSON string `{ "name": "Darth" }`:

```
curl -d '{ "name": "Darth" }' http://example.com
```

Or if you want to avoid the single quote thing, you may prefer to send the data to curl via a file, which then doesn't need the extra quoting. Assuming we call the file 'json' that contains the above mentioned data:

```
curl -d @json http://example.com
```

Negative options

For options that switch on something, there is also a way to switch it off. You then use the long form of the option with an initial "no-" prefix before the name. As an example, to switch off verbose mode:

```
curl --no-verbose http://example.com
```

Options depend on version

`curl` was first typed on a command line back in the glorious year of 1998. It already then worked on the specified URL and none, one or more command-line options given to it.

Since then we have added more options. We add options as we go along and almost every new release of curl has one or a few new options that allow users to modify certain aspects of its operation.

With the curl project's rather speedy release chain with a new release shipping every eight weeks, it is almost inevitable that you are at least not always using the very latest released version of curl. Sometimes you may even use a curl version that is a few years old.

All command-line options described in this book were, of course, added to curl at some point in time, and only a very small portion of them were available that fine spring day in 1998 when curl first shipped. You may have reason to check your version of curl and crosscheck with the curl man page for when certain options were added. This is especially important if you want to take a curl command line using a modern curl version back to an older system that might be running an older installation.

The developers of curl are working hard to not change existing behavior though. Command lines written to use curl in 1998, 2003 or 2010 should all be possible to run unmodified even today.

URLs

curl is called curl because a substring in its name is URL (Uniform Resource Locator). It operates on URLs. URL is the name we casually use for the web address strings, like the ones we usually see prefixed with http:// or starting with www.

URL is, strictly speaking, the former name for these. URI (Uniform Resource Identifier) is the more modern and correct name for them. Their syntax is defined in RFC 3986.

Where curl accepts a "URL" as input, it is then really a "URI". Most of the protocols curl understands also have a corresponding URI syntax document that describes how that particular URI format works.

curl assumes that you give it a valid URL and it only does limited checks of the format in order to extract the information it deems necessary to perform its operation. You can, for example, most probably pass in illegal characters in the URL without curl noticing or caring and it will just pass them on.

Scheme

URLs start with the "scheme", which is the official name for the "http://" part. That tells which protocol the URL uses. The scheme must be a known one that this version of curl supports or it will show an error message and stop. Additionally, the scheme must neither start with nor contain any whitespace.

The scheme separator

The scheme identifier is separated from the rest of the URL by the "://" sequence. That is a colon and two forward slashes. There exists URL formats with only one slash, but curl doesn't support any of them. There are two additional notes to be aware of, about the number of slashes:

curl allow some illegal syntax and try to correct it internally; so it will also understand and accept URLs with one or three slashes, even though they are in fact not properly formed URLs. curl does this because the browsers started this practice so it has lead to such URLs being used in the wild every now and then.

`file://` URLs are written as `file://<hostname>/<path>` but the only hostnames that are okay to use are `localhost` , `127.0.0.1` or a blank (nothing at all):

```
file://localhost/path/to/file
file://127.0.0.1/path/to/file
file:///path/to/file
```

Inserting any other host name in there will make recent versions of curl to return an error.

Pay special attention to the third example above (`file:///path/to/file`). That is *three* slashes before the path. That is again an area with common mistakes and where browsers allow users to use the wrong syntax so as a special exception, curl on Windows also allows this incorrect format:

```
file://X:/path/to/file
```

… where X is a windows-style drive letter.

Without scheme

As a convenience, curl also allows users to leave out the scheme part from URLs. Then it guesses which protocol to use based on the first part of the host name. That guessing is very basic as it just checks if the first part of the host name matches one of a set of protocols, and assumes you meant to use that protocol. This heuristic is based on the fact that servers traditionally used to be named like that. The protocols that are detected this way are FTP, DICT, LDAP, IMAP, SMTP and POP3. Any other host name in a scheme-less URL will make curl default to HTTP.

You can modify the default protocol to something other than HTTP with the `--proto-default` option.

Name and password

After the scheme, there can be a possible user name and password embedded. The use of this syntax is usually frowned upon these days since you easily leak this information in scripts or otherwise. For example, listing the directory of an FTP server using a given name and password:

```
curl ftp://user:password@example.com/
```

The presence of user name and password in the URL is completely optional. curl also allows that information to be provide with normal command-line options, outside of the URL.

Host name or address

The host name part of the URL is, of course, simply a name that can be resolved to an numerical IP address, or the numerical address itself. When specifying a numerical address, use the dotted version for IPv4 addresses:

```
curl http://127.0.0.1/
```

…and for IPv6 addresses the numerical version needs to be within square brackets:

```
curl http://[::1]/
```

When a host name is used, the converting of the name to an IP address is typically done using the system's resolver functions. That normally lets a sysadmin provide local name lookups in the `/etc/hosts` file (or equivalent).

Port number

Each protocol has a "default port" that curl will use for it, unless a specified port number is given. The optional port number can be provided within the URL after the host name part, as a colon and the port number written in decimal. For example, asking for an HTTP document on port 8080:

```
curl http://example.com:8080/
```

With the name specified as an IPv4 address:

```
curl http://127.0.0.1:8080/
```

With the name given as an IPv6 address:

```
curl http://[fdea::1]:8080/
```

Path

Every URL contains a path. If there's none given, "/" is implied. The path is sent to the specified server to identify exactly which resource that is requested or that will be provided.

The exact use of the path is protocol dependent. For example, getting a file README from the default anonymous user from an FTP server:

```
curl ftp://ftp.example.com/README
```

For the protocols that have a directory concept, ending the URL with a trailing slash means that it is a directory and not a file. Thus asking for a directory list from an FTP server is implied with such a slash:

```
curl ftp://ftp.example.com/tmp/
```

FTP type

This is not a feature that is widely used.

URLs that identify files on FTP servers have a special feature that allows you to also tell the client (curl in this case) which file type the resource is. This is because FTP is a little special and can change mode for a transfer and thus handle the file differently than if it would use another mode.

You tell curl that the FTP resource is an ASCII type by appending ";type=A" to the URL. Getting the 'foo' file from example.com's root directory using ASCII could then be made with:

```
curl "ftp://example.com/foo;type=A"
```

And while curl defaults to binary transfers for FTP, the URL format allows you to also specify the binary type with type=I:

```
curl "ftp://example.com/foo;type=I"
```

Finally, you can tell curl that the identified resource is a directory if the type you pass is D:

```
curl "ftp://example.com/foo;type=D"
```

…this can then work as an alternative format, instead of ending the path with a trailing slash as mentioned above.

Fragment

URLs offer a "fragment part". That's usually seen as a hash symbol (#) and a name for a specific name within a web page in browsers. curl supports fragments fine when a URL is passed to it, but the fragment part is never actually sent over the wire so it doesn't make a difference to curl's operations whether it is present or not.

Browsers' "address bar"

It is important to realize that when you use a modern web browser, the "address bar" they tend to feature at the top of their main windows are not using "URLs" or even "URIs". They are in fact mostly using IRIs, which is a superset of URIs to allow internationalization like non-Latin symbols and more, but it usually goes beyond that, too, as they tend to, for example, handle spaces and do magic things on percent encoding in ways none of these mentioned specifications say a client should do.

The address bar is quite simply an interface for humans to enter and see URI-like strings.

Sometimes the differences between what you see in a browser's address bar and what you can pass in to curl is significant.

Many options and URLs

As mentioned above, curl supports hundreds of command-line options and it also supports an unlimited number of URLs. If your shell or command-line system supports it, there's really no limit to how long a command line you can pass to curl.

curl will parse the entire command line first, apply the wishes from the command-line options used, and then go over the URLs one by one (in a left to right order) to perform the operations.

For some options (for example `-o` or `-O` that tell curl where to store the transfer), you may want to specify one option for each URL on the command line.

curl will return an exit code for its operation on the last URL used. If you instead rather want curl to exit with an error on the first URL in the set that fails, use the `--fail-early` option.

Separate options per URL

In previous sections we described how curl always parses all options in the whole command line and applies those to all the URLs that it transfers.

That was a simplification: curl also offers an option (-;, --next) that inserts a sort of boundary between a set of options and URLs for which it will apply the options. When the command-line parser finds a --next option, it applies the following options to the next set of URLs. The --next option thus works as a *divider* between a set of options and URLs. You can use as many --next options as you please.

As an example, we do an HTTP GET to a URL and follow redirects, we then make a second HTTP POST to a different URL and we round it up with a HEAD request to a third URL. All in a single command line:

```
curl --location http://example.com/1 --next
  --data sendthis http://example.com/2 --next
  --head http://example.com/3
```

Trying something like that *without* the --next options on the command line would generate an illegal command line since curl would attempt to combine both a POST and a HEAD:

```
Warning: You can only select one HTTP request method! You asked for both POST
Warning: (-d, --data) and HEAD (-I, --head).
```

Connection reuse

Setting up a TCP connection and especially a TLS connection can be a slow process, even on high bandwidth networks.

It can be useful to remember that curl has a connection pool internally which keeps previously used connections alive and around for a while after they were used so that subsequent requests to the same hosts can reuse an already established connection.

Of course, they can only be kept alive for as long as the curl tool is running, but it is a very good reason for trying to get several transfers done within the same command line instead of running several independent curl command line invocations.

URL globbing

At times you want to get a range of URLs that are mostly the same, with only a small portion of it changing between the requests. Maybe it is a numeric range or maybe a set of names. curl offers "globbing" as a way to specify many URLs like that easily.

The globbing uses the reserved symbols [] and {} for this, symbols that normally cannot be part of a legal URL (except for numerical IPv6 addresses but curl handles them fine anyway). If the globbing gets in your way, disable it with `-g, --globoff`.

While most transfer related functionality in curl is provided by the libcurl library, the URL globbing feature is not!

Numerical ranges

You can ask for a numerical range with [N-M] syntax, where N is the start index and it goes up to and including M. For example, you can ask for 100 images one by one that are named numerically:

```
curl -O http://example.com/[1-100].png
```

and it can even do the ranges with zero prefixes, like if the number is three digits all the time:

```
curl -O http://example.com/[001-100].png
```

Or maybe you only want even numbered images so you tell curl a step counter too. This example range goes from 0 to 100 with an increment of 2:

```
curl -O http://example.com/[0-100:2].png
```

Alphabetical ranges

curl can also do alphabetical ranges, like when a site has sections named a to z:

```
curl -O http://example.com/section[a-z].html
```

A list

Sometimes the parts don't follow such an easy pattern, and then you can instead give the full list yourself but then within the curly braces instead of the brackets used for the ranges:

```
curl -O http://example.com/{one,two,three,alpha,beta}.html
```

Combinations

You can use several globs in the same URL which then will make curl iterate over those, too. To download the images of Ben, Alice and Frank, in both the resolutions 100x100 and 1000x1000, a command line could look like:

```
curl -O http://example.com/{Ben,Alice,Frank}-{100x100,1000x1000}.jpg
```

Or download all the images of a chess board, indexed by two coordinates ranged 0 to 7:

```
curl -O http://example.com/chess-[0-7]x[0-7].jpg
```

And you can, of course, mix ranges and series. Get a week's worth of logs for both the web server and the mail server:

```
curl -O http://example.com/{web,mail}-log[0-6].txt
```

Output variables for globbing

In all the globbing examples previously in this chapter we have selected to use the `-O / --remote-name` option, which makes curl save the target file using the file name part of the used URL.

Sometimes that is not enough. You are downloading multiple files and maybe you want to save them in a different subdirectory or create the saved file names differently. curl, of course, has a solution for these situations as well: output file name variables.

Each "glob" used in a URL gets a separate variable. They are referenced as '#[num]' - that means the single letter '#' followed by the glob number which starts with 1 for the first glob and ends with the last glob.

Save the main pages of two different sites:

```
curl http://{one,two}.example.com -o "file_#1.txt"
```

Save the outputs from a command line with two globs in a subdirectory;

```
curl http://{site,host}.host[1-5].example.com -o "subdir/#1_#2"
```

List all command-line options

curl has more than two hundred command-line options and the number of options keep increasing over time. Chances are the number of options will reach 250 within a few years.

In order to find out which options you need to perform as certain action, you can, of course, list all options, scan through the list and pick the one you are looking for. `curl --help` or simply `curl -h` will get you a list of all existing options with a brief explanation. If you don't really know what you are looking for, you probably won't be entirely satisfied.

Then you can instead opt to use `curl --manual` which will output the entire man page for curl plus an appended tutorial for the most common use cases. That is a very thorough and complete document on how each option works amassing several thousand lines of documentation. To wade through that is also a tedious work and we encourage use of a search function through those text masses. Some people will appreciate the man page in its web version.

Config file

You can easily end up with curl command lines that use a very large number of command-line options, making them rather hard to work with. Sometimes the length of the command line you want to enter even hits the maximum length your command-line system allows. The Microsoft Windows command prompt being an example of something that has a fairly small maximum line length.

To aid such situations, curl offers a feature we call "config file". It basically allows you to write command-line options in a text file instead and then tell curl to read options from that file in addition to the command line.

You tell curl to read more command-line options from a specific file with the -K/--config option, like this:

```
curl -K cmdline.txt http://example.com
```

…and in the `cmdline.txt` file (which, of course, can use any file name you please) you enter each command line per line:

```
# this is a comment, we ask to follow redirects
--location
# ask to do a HEAD request
--head
```

The config file accepts both short and long options, exactly as you would write them on a command line. As a special extra feature, it also allows you to write the long format of the options without the leading two dashes to make it easier to read. Using that style, the config file shown above can alternatively be written as:

```
# this is a comment, we ask to follow redirects
location
# ask to do a HEAD request
head
```

Command line options that take an argument must have its argument provided on the same line as the option. For example changing the User-Agent HTTP header can be done with

```
user-agent "Everything-is-an-agent"
```

To allow the config files to look even more like a true config file, it also allows you to use '=' or ':' between the option and its argument. As you see above it isn't necessary, but some like the clarity it offers. Setting the user-agent option again:

```
user-agent = "Everything-is-an-agent"
```

The argument to an option can be specified without double quotes and then curl will treat the next space or newline as the end of the argument. So if you want to provide an argument with embedded spaces you must use double quotes.

The user agent string example we have used above has no white spaces and therefore it can also be provided without the quotes like:

```
user-agent = Everything-is-an-agent
```

Finally, if you want to provide a URL in a config file, you must do that the `--url` way, or just with `url`, and not like on the command line where basically everything that isn't an option is assumed to be a URL. So you provide a URL for curl like this:

```
url = "http://example.com"
```

Default config file

When curl is invoked, it always (unless `-q` is used) checks for a default config file and uses it if found. The file name it checks for is `.curlrc` on Unix-like systems and `_curlrc` on Windows.

The default config file is checked for in the following places in this order:

1. curl tries to find the "home directory": It first checks for the CURL_HOME and then the HOME environment variables. Failing that, it uses `getpwuid()` on Unix-like systems (which returns the home directory given the current user in your system). On Windows, it then checks for the APPDATA variable, or as a last resort the '%USERPROFILE%\Application Data'.
2. On Windows, if there is no _curlrc file in the home directory, it checks for one in the same directory the curl executable is placed. On Unix-like systems, it will simply try to load .curlrc from the determined home directory.

Passwords and snooping

Passwords are tricky and sensitive. Leaking a password can make someone else than you access the resources and the data otherwise protected.

curl offers several ways to receive passwords from the user and then subsequently pass them on or use them to something else.

The most basic curl authentication option is `-u / --user`. It accepts an argument that is the user name and password, colon separated. Like when alice wants to request a page requiring HTTP authentication and her password is '12345':

$ curl -u alice:12345 http://example.com/

Command line leakage

Several potentially bad things are going on here. First, we are entering a password on the command line and the command line might be readable for other users on the same system (assuming you have a multi-user system). curl will help minimize that risk by trying to blank out passwords from process listings.

One way to avoid passing the user name and password on the command line is to instead use a .netrc file or a config file. You can also use the `-u` option without specifying the password, and then curl will instead prompt the user for it when it runs.

Network leakage

Secondly, this command line sends the user credentials to an HTTP server, which is a clear-text protocol that is open for man-in-the-middle or other snoopers to spy on the connection and see what is sent. In this command line example, it makes curl use HTTP Basic authentication and that is completely insecure.

There are several ways to avoid this, and the key is, of course, then to avoid protocols or authentication schemes that sends credentials in the plain over the network. Easiest is perhaps to make sure you use encrypted versions of protocols. Use HTTPS instead of HTTP, use FTPS instead of FTP and so on.

If you need to stick to a plain text and insecure protocol, then see if you can switch to using an authentication method that avoids sending the credentials in the clear. If you want HTTP, such methods would include Digest (`--digest`), Negotiate (`--negotiate.`) and NTLM (`--ntlm`).

The progress meter

curl has a built-in progress meter. When curl is invoked to transfer data (either uploading or downloading) it can show that meter in the terminal screen to show how the transfer is progressing, namely the current transfer speed, how long it has been going on and how long it thinks it might be left until completion.

The progress meter is inhibited if curl deems that there is output going to the terminal, as then would the progress meter interfere with that output and just mess up what gets displayed. A user can also forcibly switch off the progress meter with the `-s / --silent` option, which tells curl to hush.

If you invoke curl and don't get the progress meter, make sure your output is directed somewhere other than the terminal.

curl also features an alternative and simpler progress meter that you enable with `-# / --progress-bar`. As the long name implies, it instead shows the transfer as progress bar.

At times when curl is asked to transfer data, it can't figure out the total size of the requested operation and that then subsequently makes the progress meter contain fewer details and it cannot, for example, make forecasts for transfer times, etc.

Units

The progress meter displays bytes and bytes per second.

It will also use suffixes for larger amounts of bytes, using the 1024 base system so 1024 is one kilobyte (1K), 2048 is 2K, etc. curl supports these:

Suffix	Amount	Name
K	2^10	kilobyte
M	2^20	megabyte
G	2^30	gigabyte
T	2^40	terabyte
P	2^50	petabyte

The times are displayed using H:MM:SS for hours, minutes and seconds.

Progress meter legend

The progress meter exists to show a user that something actually is happening. The different fields in the output have the following meaning:

```
% Total    % Received % Xferd  Average Speed          Time             Curr.
                               Dload  Upload Total    Current  Left    Speed
0  151M    0 38608    0     0   9406      0  4:41:43  0:00:04  4:41:39  9287
```

From left to right:

Title	Meaning
%	Percentage completed of the whole transfer
Total	Total size of the whole expected transfer (if known)
%	Percentage completed of the download
Received	Currently downloaded number of bytes
%	Percentage completed of the upload
Xferd	Currently uploaded number of bytes
Average Speed Dload	Average transfer speed of the entire download so far, in number of bytes per second
Average Speed Upload	Average transfer speed of the entire upload so far, in number of bytes per second
Time Total	Expected time to complete the operation, in HH:MM:SS notation for hours, minutes and seconds
Time Current	Time passed since the start of the transfer, in HH:MM:SS notation for hours, minutes and seconds
Time Left	Expected time left to completion, in HH:MM:SS notation for hours, minutes and seconds
Curr.Speed	Average transfer speed over the last 5 seconds (the first 5 seconds of a transfer is based on less time, of course) in number of bytes per second

Using curl

Previous chapters have described some basic details on what curl is and something about the basic command lines. You use command-line options and you pass on URLs to work with.

In this chapter, we are going to dive deeper into a variety of different concepts of what curl can do and how to tell curl to use these features. You should consider all these features as different tools that are here to help you do your file transfer tasks as conveniently as possible.

Supported protocols

curl supports or can be made to support (if built so) the following protocols.

DICT, FILE, FTP, FTPS, GOPHER, HTTP, HTTPS, IMAP, IMAPS, LDAP, LDAPS, POP3, POP3S, RTMP, RTSP, SCP, SFTP, SMB, SMTP, SMTPS, TELNET and TFTP

Verbose mode

If your curl command doesn't execute or return what you expected it to, your first gut reaction should always be to run the command with the `-v / --verbose` option to get more information.

When verbose mode is enabled, curl gets more talkative and will explain and show a lot more of its doings. It will add informational tests and prefix them with '*'. For example, let's see what curl might say when trying a simple HTTP example (saving the downloaded data in the file called 'saved'):

```
$ curl -v http://example.com -o saved
* Rebuilt URL to: http://example.com/
```

Ok so we invoked curl with a URL that it considers incomplete so it helps us and it adds a trailing slash before it moves on.

```
*   Trying 93.184.216.34...
```

This tells us curl now tries to connect to this IP address. It means the name 'example.com' has been resolved to one or more addresses and this is the first (and possibly only) address curl will try to connect to.

```
* Connected to example.com (93.184.216.34) port 80 (#0)
```

It worked! curl connected to the site and here it explains how the name maps to the IP address and on which port it has connected to. The '(#0)' part is which internal number curl has given this connection. If you try multiple URLs in the same command line you can see it use more connections or reuse connections, so the connection counter may increase or not increase depending on what curl decides it needs to do.

If we use an HTTPS:// URL instead of an HTTP one, there will also be a whole bunch of lines explaining how curl uses CA certs to verify the server's certificate and some details from the server's certificate, etc. Including which ciphers were selected and more TLS details.

In addition to the added information given from curl internals, the -v verbose mode will also make curl show all headers it sends and receives. For protocols without headers (like FTP, SMTP, POP3 and so on), we can consider commands and responses as headers and they will thus also be shown with -v.

If we then continue the output seen from the command above (but ignore the actual HTML response), curl will show:

```
> GET / HTTP/1.1
> Host: example.com
> User-Agent: curl/7.45.0
> Accept: */*
>
```

This is the full HTTP request to the site. This request is how it looks in a default curl 7.45.0 installation and it may, of course, differ slightly between different releases and in particular it will change if you add command line options.

The last line of the HTTP request headers looks empty, and it is. It signals the separation between the headers and the body, and in this request there is no "body" to send.

Moving on and assuming everything goes according to plan, the sent request will get a corresponding response from the server and that HTTP response will start with a set of headers before the response body:

```
< HTTP/1.1 200 OK
< Accept-Ranges: bytes
< Cache-Control: max-age=604800
< Content-Type: text/html
< Date: Sat, 19 Dec 2015 22:01:03 GMT
< Etag: "359670651"
< Expires: Sat, 26 Dec 2015 22:01:03 GMT
< Last-Modified: Fri, 09 Aug 2013 23:54:35 GMT
< Server: ECS (ewr/15BD)
< Vary: Accept-Encoding
< X-Cache: HIT
< x-ec-custom-error: 1
< Content-Length: 1270
<
```

This may look mostly like mumbo jumbo to you, but this is normal set of HTTP headers—metadata—about the response. The first line's "200" might be the most important piece of information in there and means "everything is fine".

The last line of the received headers is, as you can see, empty, and that is the marker used for the HTTP protocol to signal the end of the headers.

After the headers comes the actual response body, the data payload. The regular -v verbose mode does not show that data but only displays

```
{ [1270 bytes data]
```

That 1270 bytes should then be in the 'saved' file. You can also see that there was a header named Content-Length: in the response that contained the exact file length (it won't always be present in responses).

--trace and --trace-ascii

There are times when `-v` is not enough. In particular, when you want to store the complete stream including the actual transferred data.

For situations when curl does encrypted file transfers with protocols such as HTTPS, FTPS or SFTP, other network monitoring tools (like Wireshark or tcpdump) won't be able to do this job as easily for you.

For this, curl offers two other options that you use instead of `-v`.

`--trace [filename]` will save a full trace in the given file name. You can also use '-' (a single minus) instead of a file name to get it passed to stdout. You would use it like this:

```
$ curl --trace dump http://example.com
```

When completed, there's a 'dump' file that can turn out pretty sizable. In this case, the 15 first lines of the dump file looks like:

```
== Info: Rebuilt URL to: http://example.com/
== Info:   Trying 93.184.216.34...
== Info: Connected to example.com (93.184.216.34) port 80 (#0)
=> Send header, 75 bytes (0x4b)
0000: 47 45 54 20 2f 20 48 54 54 50 2f 31 2e 31 0d 0a GET / HTTP/1.1..
0010: 48 6f 73 74 3a 20 65 78 61 6d 70 6c 65 2e 63 6f Host: example.co
0020: 6d 0d 0a 55 73 65 72 2d 41 67 65 6e 74 3a 20 63 m..User-Agent: c
0030: 75 72 6c 2f 37 2e 34 35 2e 30 0d 0a 41 63 63 65 url/7.45.0..Acce
0040: 70 74 3a 20 2a 2f 2a 0d 0a 0d 0a                pt: */*....
<= Recv header, 17 bytes (0x11)
0000: 48 54 54 50 2f 31 2e 31 20 32 30 30 20 4f 4b 0d HTTP/1.1 200 OK.
0010: 0a                                              .
<= Recv header, 22 bytes (0x16)
0000: 41 63 63 65 70 74 2d 52 61 6e 67 65 73 3a 20 62 Accept-Ranges: b
0010: 79 74 65 73 0d 0a                               ytes..
```

Every single sent and received byte get displayed individually in hexadecimal numbers.

If you think the hexadecimals aren't helping, you can try `--trace-ascii [filename]` instead, also this accepting '-' for stdout and that makes the 15 first lines of tracing look like:

```
== Info: Rebuilt URL to: http://example.com/
== Info:   Trying 93.184.216.34...
== Info: Connected to example.com (93.184.216.34) port 80 (#0)
=> Send header, 75 bytes (0x4b)
0000: GET / HTTP/1.1
0010: Host: example.com
0023: User-Agent: curl/7.45.0
003c: Accept: */*
0049:
<= Recv header, 17 bytes (0x11)
0000: HTTP/1.1 200 OK
<= Recv header, 22 bytes (0x16)
0000: Accept-Ranges: bytes
<= Recv header, 31 bytes (0x1f)
0000: Cache-Control: max-age=604800
```

--trace-time

This options prefixes all verbose/trace outputs with a high resolution timer for when the line is printed. It works with the regular `-v / --verbose` option as well as with `--trace` and `--trace-ascii`.

An example could look like this:

```
$ curl -v --trace-time http://example.com
23:38:56.837164 * Rebuilt URL to: http://example.com/
23:38:56.841456 *   Trying 93.184.216.34...
23:38:56.935155 * Connected to example.com (93.184.216.34) port 80 (#0)
23:38:56.935296 > GET / HTTP/1.1
23:38:56.935296 > Host: example.com
23:38:56.935296 > User-Agent: curl/7.45.0
23:38:56.935296 > Accept: */*
23:38:56.935296 >
23:38:57.029570 < HTTP/1.1 200 OK
23:38:57.029699 < Accept-Ranges: bytes
23:38:57.029803 < Cache-Control: max-age=604800
23:38:57.029903 < Content-Type: text/html
---- snip ----
```

The lines are all the local time as hours:minutes:seconds and then number of microseconds in that second.

HTTP/2

When doing file transfers using version two of the HTTP protocol, HTTP/2, curl sends and receives **compressed** headers. So to display outgoing and incoming HTTP/2 headers in a readable and understandable way, curl will actually show the uncompressed versions in a style similar to how they appear with HTTP/1.1.

--write-out

This is one of the often forgotten little gems in the curl arsenal of command line options. `--write-out` or just `-w` for short, writes out information after a transfer has completed and it has a large range of variables that you can include in the output, variables that have been set with values and information from the transfer.

You tell curl to write a string just by passing that string to this option:

```
curl -w "formatted string" http://example.com/
```

…and you can also have curl read that string from a given file instead if you prefix the string with '@':

```
curl -w @filename http://example.com/
```

…or even have curl read the string from stdin if you use '-' as filename:

```
curl -w @- http://example.com/
```

The variables that are available are accessed by writing `%{variable_name}` in the string and that variable will then be substituted by the correct value. To output a normal '%' you just write it as '%%'. You can also output a newline by using '\n', a carriage return with '\r' and a tab space with '\t'.

(The %-symbol is special on the Windows command line, where all occurrences of % must be doubled when using this option.)

As an example, we can output the Content-Type and the response code from an HTTP transfer, separated with newlines and some extra text like this:

```
curl -w "Type: %{content_type}\nCode: %{response_code}\n" http://example.com
```

This feature writes the output to stdout so you probably want to make sure that you don't also send the downloaded content to stdout as then you might have a hard time to separate out the data.

Available --write-out variables

Some of these variables are not available in really old curl versions.

- %{content_type} shows the Content-Type of the requested document, if there was any.
- %{filename_effective} shows the ultimate filename that curl writes out to. This is only meaningful if curl is told to write to a file with the `--remote-name` or `--output` option. It's most useful in combination with the `--remote-header-name` option.
- %{ftp_entry_path} shows the initial path curl ended up in when logging on to the remote FTP server.
- %{response_code} shows the numerical response code that was found in the last transfer.
- %{http_connect} shows the numerical code that was found in the last response (from a proxy) to a curl CONNECT request.
- %{local_ip} shows the IP address of the local end of the most recently done connection—can be either IPv4 or IPv6
- %{local_port} shows the local port number of the most recently made connection
- %{num_connects} shows the number of new connects made in the recent transfer.
- %{num_redirects} shows the number of redirects that were followed in the request.
- %{redirect_url} shows the actual URL a redirect *would* take you to when an HTTP request was made without `-L` to follow redirects.
- %{remote_ip} shows the remote IP address of the most recently made connection—can be either IPv4 or IPv6.
- %{remote_port} shows the remote port number of the most recently made connection.
- %{size_download} shows the total number of bytes that were downloaded.
- %{size_header} shows the total number of bytes of the downloaded headers.
- %{size_request} shows the total number of bytes that were sent in the HTTP request.
- %{size_upload} shows the total number of bytes that were uploaded.
- %{speed_download} shows the average download speed that curl measured for the complete download in bytes per second.
- %{speed_upload} shows the average upload speed that curl measured for the complete upload in bytes per second.

- %{ssl_verify_result} shows the result of the SSL peer certificate verification that was requested. 0 means the verification was successful.
- %{time_appconnect} shows the time, in seconds, it took from the start until the SSL/SSH/etc connect/handshake to the remote host was completed.
- %{time_connect} shows the time, in seconds, it took from the start until the TCP connect to the remote host (or proxy) was completed.
- %{time_namelookup} shows the time, in seconds, it took from the start until the name resolving was completed.
- %{time_pretransfer} shows the time, in seconds, it took from the start until the file transfer was just about to begin. This includes all pre-transfer commands and negotiations that are specific to the particular protocol(s) involved.
- %{time_redirect} shows the time, in seconds, it took for all redirection steps including name lookup, connect, pre-transfer and transfer before the final transaction was started. time_redirect shows the complete execution time for multiple redirections.
- %{time_starttransfer} shows the time, in seconds, it took from the start until the first byte was just about to be transferred. This includes time_pretransfer and also the time the server needed to calculate the result.
- %{time_total} shows the total time, in seconds, that the full operation lasted. The time will be displayed with millisecond resolution.
- %{url_effective} shows the URL that was fetched last. This is particularly meaningful if you have told curl to follow Location: headers (with `-L`).

Silence

The opposite of verbose is, of course, to make curl more silent. With the `-s` (or `--silent`) option you make curl switch off the progress meter and not output any error messages for when errors occur. It gets mute. It will still output the downloaded data you ask it to.

With silence activated, you can ask for it to still output the error message on failures by adding `-S` or `--show-error`.

Persistent connections

When setting up TCP connections to sites, curl will keep the old connection around for a while so that if the next transfer is to the same host it can reuse the same connection again and thus save a lot of time. We call this persistent connections. curl will always try to keep connections alive and reuse existing connections as far as it can.

The curl command-line tool can, however, only keep connections alive for as long as it runs, so as soon as it exits back to your command line it has to close down all currently open connections (and also free and clean up all the other caches it uses to decrease time of subsequent operations). We call the pool of alive connections the "connection cache".

If you want to perform N transfers or operations against the same host or same base URL, you could gain a lot of speed by trying to do them in as few curl command lines as possible instead of repeatedly invoking curl with one URL at a time.

Downloads

"Download" means getting data from a server on a network, and the server is then clearly considered to be "above" you. This is loading data down from the server onto your machine where you are running curl.

Downloading is probably the most common use case for curl—retrieving the specific data pointed to by a URL onto your machine.

What exactly is downloading?

You specify the resource to download by giving curl a URL. curl defaults to downloading a URL unless told otherwise, and the URL identifies what to download. In this example the URL to download is "http://example.com":

```
curl http://example.com
```

The URL is broken down into its individual components (as explained elsewhere), the correct server is contacted and is then asked to deliver the specific resource—often a file. The server then delivers the data, or it refuses or perhaps the client asked for the wrong data and then that data is delivered.

A request for a resource is protocol-specific so a FTP:// URL works differently than an HTTP:// URL or an SFTP:// URL.

A URL without a path part, that is a URL that has a host name part only (like the "http://example.com" example above) will get a slash ('/') appended to it internally and then that is the resource curl will ask for from the server.

If you specify multiple URLs on the command line, curl will download each URL one by one. It won't start the second transfer until the first one is complete, etc.

Storing downloads

If you try the example download as in the previous section, you will notice that curl will output the downloaded data to stdout unless told to do something else. Outputting data to stdout is really useful when you want to pipe it into another program or similar, but it is not always the optimal way to deal with your downloads.

Give curl a specific file name to save the download in with `-o [filename]` (with `--output` as the long version of the option), where filename is either just a file name, a relative path to a file name or a full path to the file.

Also note that you can put the `-o` before or after the URL; it makes no difference:

```
curl -o output.html http://example.com/
curl -o /tmp/index.html http://example.com/
curl http://example.com -o ../../folder/savethis.html
```

This is, of course, not limited to http:// URLs but works the same way no matter which type of URL you download:

```
curl -o file.txt ftp://example.com/path/to/file-name.ext
```

If you ask curl to send the output to the terminal, it attempts to detect and prevent binary data from being sent there since that can seriously mess up your terminal (sometimes to the point where it basically stops working). You can override curl's binary-output-prevention and force the output to get sent to stdout by using `-o -`.

curl has several other ways to store and name the downloaded data. Details follow!

Download to a file named by the URL

Many URLs, however, already contain the file name part in the rightmost end. curl lets you use that as a shortcut so you don't have to repeat it with `-o`. So instead of:

```
curl -o file.html http://example.com/file.html
```

You can save the remove URL resource into the local file 'file.html' with this:

```
curl -O http://example.com/file.html
```

This is the `-O` (uppercase letter o) option, or `--remote-name` for the long name version. The -O option selects the local file name to use by picking the file name part of the URL that you provide. This is important. You specify the URL and curl picks the name from this data. If the site redirects curl further (and if you tell curl to follow redirects), it doesn't change the file name curl will use for storing this.

Get the target file name from the server

HTTP servers have the option to provide a header named `Content-Disposition:` in responses. That header may contain a suggested file name for the contents delivered, and curl can be told to use that hint to name its local file. The `-J / --remote-header-name` enables this. If you also use the `-O` option, it makes curl use the file name from the URL by default and only *if* there's actually a valid Content-Disposition header available, it switches to saving using that name.

-J has some problems and risks associated with it that users need to be aware of:

1. It will only use the rightmost part of the suggested file name, so any path or directories the server suggests will be stripped out.
2. Since the file name is entirely selected by the server, curl will, of course, overwrite any preexisting local file in your current directory if the server happens to provide such a file name.
3. File name encoding and character sets issues. curl does not decode the name in any way, so you may end up with a URL-encoded file name where a browser would otherwise decode it to something more readable using a sensible character set.

HTML and charsets

curl will download the exact binary data that the server sends. This might be of importance to you in case, for example, you download a HTML page or other text data that uses a certain character encoding that your browser then displays as expected. curl will then not translate the arriving data.

A common example where this causes some surprising results is when a user downloads a web page with something like:

```
curl https://example.com/ -o storage.html
```

…and when inspecting the `storage.html` file after the fact, the user realizes that one or more characters look funny or downright wrong. This can then very well be because the server sent the characters using charset X, while your editor and environment use charset Y. In an ideal world, we would all use UTF-8 everywhere but unfortunately, that is still not the case.

A common work-around for this issue that works decently is to use the common `iconv` utility to translate a text file to and from different charsets.

Compression

curl allows you to ask HTTP and HTTPS servers to provide compressed versions of the data and then perform automatic decompression of it on arrival. In situations where bandwidth is more limited than CPU this will help you receive more data in a shorter amount of time.

HTTP compression can be done using two different mechanisms, one which might be considered "The Right Way" and the other that is the way that everyone actually uses and is the widespread and popular way to do it! The common way to compress HTTP content is using the **Content-Encoding** header. You ask curl to use this with the `--compressed` option:

```
curl --compressed http://example.com/
```

With this option enabled (and if the server supports it) it delivers the data in a compressed way and curl will decompress it before saving it or sending it to stdout. This usually means that as a user you don't really see or experience the compression other than possibly noticing a faster transfer.

The `--compressed` option asks for Content-Encoding compression using one of the supported compression algorithms. There's also the rarer **Transfer-Encoding** method, which is the header that was created for this automated method but was never really widely adopted. You can tell curl to ask for Transfer-Encoded compression with `--tr-encoding`:

```
curl --tr-encoding http://example.com/
```

In theory, there's nothing that prevents you from using both in the same command line, although in practice, you may then experience that some servers get a little confused when ask to compress in two different ways. It's generally safer to just pick one.

Shell redirects

When you invoke curl from a shell or some other command-line prompt system, that environment generally provides you with a set of output redirection abilities. In most Linux and Unix shells and with Windows' command prompts, you direct stdout to a file with `> filename`. Using this, of course, makes the use of -o or -O superfluous.

```
curl http://example.com/ > example.html
```

Redirecting output to a file redirects all output from curl to that file, so even if you ask to transfer more than one URL to stdout, redirecting the output will get all the URLs' output stored in that single file.

```
curl http://example.com/1 http://example.com/2 > files
```

Unix shells usually allow you to redirect the *stderr* stream separately. The stderr stream is usually a stream that also gets shown in the terminal, but you can redirect it separately from the stdout stream. The stdout stream is for the data while stderr is metadata and errors, etc., that aren't data. You can redirect stderr with `2>file` like this:

```
curl http://example.com > files.html 2>errors
```

Multiple downloads

As curl can be told to download many URLs in a single command line, there are, of course, times when you want to store these downloads in nicely-named local files.

The key to understanding this is that each download URL needs its own "storage instruction". Without said "storage instruction", curl will default to sending the data to stdout. If you ask for two URLs and only tell curl where to save the first URL, the second one is sent to stdout. Like this:

```
curl -o one.html http://example.com/1 http://example.com/2
```

The "storage instructions" are read and handled in the same order as the download URLs so they don't have to be next to the URL in any way. You can round up all the output options first, last or interleaved with the URLs. You choose!

These examples all work the same way:

```
curl -o 1.txt -o 2.txt http://example.com/1 http://example.com/2
curl http://example.com/1 http://example.com/2 -o 1.txt -o 2.txt
curl -o 1.txt http://example.com/1 http://example.com/2 -o 2.txt
curl -o 1.txt http://example.com/1 -o 2.txt http://example.com/2
```

The `-O` is similarly just an instruction for a single download so if you download multiple URLs, use more of them:

```
curl -O -O http://example.com/1 http://example.com/2
```

Use the URL's file name part for all URLs

As a reaction to adding a hundred `-O` options when using a hundred URLs, we introduced an option called `--remote-name-all`. This makes `-O` the default operation for all given URLs. You can still provide individual "storage instructions" for URLs but if you leave one out for a URL that gets downloaded, the default action is then switched from stdout to -O style.

"My browser shows something else"

A very common use case is using curl to get a URL that you can get in your browser when you paste the URL in the browser's address bar.

But a browser getting a URL does so much more and in so many different ways than curl that what curl shows in your terminal output is probably not at all what you see in your browser window.

Client differences

Curl only gets exactly what you ask it to get and it never parses the actual content—the data—that the server delivers. A browser gets data and it activates different parsers depending on what kind of content it thinks it gets. For example, if the data is HTML it will parse it to display a web page and possibly download other sub resources such as images, JavaScript and CSS files. When curl downloads a HTML it will just get that single HTML resource, even if it, when parsed by a browser, would trigger a whole busload of more downloads. If you want curl to download any sub-resources as well, you need to pass those URLs to curl and ask it to get those, just like any other URLs.

Clients also differ in how they send their requests, and some aspects of a request for a resource include, for example, format preferences, asking for compressed data, or just telling the server from which previous page we are "coming from". curl's requests will differ a little or a lot from how your browser sends its requests.

Server differences

The server that receives the request and delivers data is often setup to act in certain ways depending on what kind of client it thinks communicates with it. Sometimes it is as innocent as trying to deliver the best content for the client, sometimes it is to hide some content for some clients or even to try to work around known problems in specific browsers. Then there's also, of course, various kind of login systems that might rely on HTTP authentication or cookies or the client being from the pre-validated IP address range.

Sometimes getting the same response from a server using curl as the response you get with a browser ends up really hard work. Users then typically record their browser sessions with the browser's networking tools and then compare that recording with recorded data from

curl's `--trace-ascii` option and proceed to modify curl's requests (often with `-H / --header`) until the server starts to respond the same to both.

This type of work can be both time consuming and tedious. You should always do this with permission from the server owners or admins.

Intermediaries' fiddlings

Intermediaries are proxies, explicit or implicit ones. Some environments will force you to use one or you may choose to use one for various reasons, but there are also the transparent ones that will intercept your network traffic silently and proxy it for you no matter what you want.

Proxies are "middle men" that terminate the traffic and then act on your behalf to the remote server. This can introduce all sorts of explicit filtering and "saving" you from certain content or even "protecting" the remote server from what data you try to send to it, but even more so it introduces another software's view on how the protocol works and what the right things to do are.

Interfering intermediaries are often the cause of lots of head aches and mysteries down to downright malicious modifications of content.

We strongly encourage you to use HTTPS or other means to verify that the contents you are downloading or uploading are really the data that the remote server has sent to you and that your precious bytes end up verbatim at the intended destination.

Rate limiting

When curl transfers data, it will attempt to do that as fast as possible. It goes for both uploads and downloads. Exactly how fast that will be depends on several factors, including your computer's ability, your own network connection's bandwidth, the load on the remote server you are transferring to/from and the latency to that server. And your curl transfers are also likely to compete with other transfers on the networks the data travels over, from other users or just other apps by the same user.

In many setups, however, you will find that you can more or less saturate your own network connection with a single curl command line. If you have a 10 megabit per second connection to the Internet, chances are curl can use all of those 10 megabits to transfer data.

For most use cases, using as much bandwidth as possible is a good thing. It makes the transfer faster, it makes the curl command complete sooner and it will make the transfer use resources from the server for a shorter period of time.

Sometimes you will, however, find that having curl starve out other network functions on your local network connection is inconvenient. In these situations you may want to tell curl to slow down so that other network users get a better chance to get their data through as well. With `--limit-rate [speed]` you can tell curl to not go faster than the given number of bytes per second. The rate limit value can be given with a letter suffix using one of K, M and G for kilobytes, megabytes and gigabytes.

To make curl not download data any faster than 200 kilobytes per second:

```
curl https://example.com/ --limit-rate 200K
```

The given limit is the maximum *average speed* allowed, counted during the entire transfer. It means that curl might use higher transfer speeds in short bursts, but over time it uses no more than the given rate.

Also note that curl never knows what the maximum possible speed is—it will simply go as fast as it can and is allowed. You may know your connection's maximum speed, but curl does not.

Maximum filesize

When you want to make sure your curl command line won't try to download a too-large file, you can instruct curl to stop before doing that, if it knows the size before the transfer starts! Maybe that would use too much bandwidth, take too long time or you don't have enough space on your hard drive:

```
curl --max-filesize 100000 https://example.com/
```

Give curl the largest download you will accept in number of bytes and if curl can figure out the size before the transfer starts it will abort before trying to download something larger.

There are many situations in which curl cannot figure out the size at the time the transfer starts and this option will not affect those transfers, even if they may end up larger than the specified amount.

Metalink

Metalink is a file description standard that tells a client multiple locations where the same content resides. A client can then opt to transfer that content from one or many of those sources.

curl supports the Metalink format when asked to with the `--metalink` option. Then given URL should then point to a Metalink file. Such as:

```
curl --metalink https://example.com/example.metalink
```

curl will make use of the mirrors listed within the file for failover if there are errors (such as the file or server not being available). It will also verify the hash of the file after the download completes. The Metalink file itself is downloaded and processed in memory and not stored in the local file system.

Storing metadata in file system

When saving a download to a file with curl, the `--xattr` option tells curl to also store certain file metadata in "extended file attributes". These extended attributes are basically standardized name/value pairs stored in the file system, assuming one of the supported file systems and operating systems are used.

Currently, the URL is stored in the `xdg.origin.url` attribute and, for HTTP, the content type is stored in the `mime_type` attribute. If the file system does not support extended attributes when this option is set, a warning is issued.

Raw

When `--raw` is used, it disables all internal HTTP decoding of content or transfer encodings and instead makes curl passed on unaltered, raw, data.

This is typically used if you are writing some sort of middle software and you want to pass on the content to perhaps another HTTP client and allow that to do the decoding instead.

Retrying failed attempts

Normally curl will only make a single attempt to perform a transfer and return an error if not successful. Using the `--retry` option you can tell curl to retry certain failed transfers.

If a transient error is returned when curl tries to perform a transfer, it will retry this number of times before giving up. Setting the number to 0 makes curl do no retries (which is the default). Transient error means either: a timeout, an FTP 4xx response code or an HTTP 5xx response code.

When curl is about to retry a transfer, it will first wait one second and then for all forthcoming retries it will double the waiting time until it reaches 10 minutes which then will be the delay between the rest of the retries. Using `--retry-delay` you can disable this exponential

backoff algorithm and set your own delay between the attempts. With `--retry-max-time` you cap the total time allowed for retries. The `--max-time` option will still specify the longest time a single of these transfers is allowed to spend.

Resuming and ranges

Resuming a download means first checking the size of what is already present locally and then asking the server to send the rest of it so it can be appended. curl also allows resuming the transfer at a custom point without actually having anything already locally present.

curl supports resumed downloads on several protocols. Tell it where to start the transfer with the `-C, --continue-at` option that takes either a plain numerical byte counter offset where to start or the string `-` that asks curl to figure it out itself based on what it knows. When using `-`, curl will use the destination file name to figure out how much data that is already present locally and ask use that as an offset when asking for more data from the server.

To start downloading an FTP file from byte offset 100:

```
curl --continue-at 100 ftp://example.com/bigfile
```

Continue downloading a previously interrupted download:

```
curl --continue-at - http://example.com/bigfile -O
```

If you instead just want a specific byte range from the remote resource transferred, you can ask for only that. For example, when you only want 1000 bytes from offset 100 to avoid having to download the entire huge remote file:

```
curl --range 100-1999 http://example.com/bigfile
```

Uploads

Uploading is a term for sending data to a remote server. Uploading is done differently for each protocol, and several protocols may even allow different ways of uploading data.

Protocols allowing upload

You can upload data using one of these protocols: FILE, FTP, FTPS, HTTP, HTTPS, IMAP, IMAPS, SCP, SFTP, SMB, SMBS, SMTP, SMTPS and TFTP.

HTTP offers several "uploads"

HTTP (and its bigger brother HTTPS) provides several different ways to upload data to a server and curl offers easy command-line options to do it the three most common ways, described below.

An interesting detail with HTTP is also that an upload can also be a download, in the same operation and in fact many downloads are initiated with an HTTP POST.

POST

POST is the HTTP method that was invented to send data to a receiving web application, and it is, for example, how most common HTML forms on the web work. It usually sends a chunk of relatively small amounts of data to the receiver.

The upload kind is usually done with the `-d` or `--data` options, but there are a few additional alterations.

Read the detailed description on how to do this with curl in the HTTP POST with curl chapter.

multipart formpost

Multipart formposts are also used in HTML forms on web sites; typically when there's a file upload involved. This type of upload is also an HTTP POST but it sends the data formatted according to some special rules, which is what the "multipart" name means.

Since it sends the data formatted completely differently, you cannot select which type of POST to use at your own whim but it entirely depends on what the receiving server end expects and can handle.

HTTP multipart formposts are done with `-F`. See the detailed description in the HTTP multipart formposts chapter.

PUT

HTTP PUT is the sort of upload that was designed to send a complete resource that is meant to be put as-is on the remote site or even replace an existing resource there. That said, this is also the least used upload method for HTTP on the web today and lots, if not most, web servers don't even have PUT enabled.

You send off an HTTP upload using the -T option with the file to upload:

```
curl -T uploadthis http://example.com/
```

FTP uploads

Working with FTP, you get to see the remote file system you will be accessing. You tell the server exactly in which directory you want the upload to be placed and which file name to use. If you specify the upload URL with a trailing slash, curl will append the locally used file name to the URL and then that will be the file name used when stored remotely:

```
curl -T uploadthis ftp://example.com/this/directory/
```

So if you prefer to select a different file name on the remote side than what you have used locally, you specify it in the URL:

```
curl -T uploadthis ftp://example.com/this/directory/remotename
```

Learn more about FTPing with curl in the Using curl/FTP section.

SMTP uploads

You may not consider sending an e-mail to be "uploading", but to curl it is. You upload the mail body to the SMTP server. With SMTP, you also need to include all the e-mail headers you need (To:, From:, Date:, etc.) in the mail body as curl will not add any at all.

```
curl -T mail smtp://mail.example.com/ --mail-from user@example.com
```

Learn more about using SMTP with curl in the Using curl/SMTP section.

Progress meter for uploads

The general progress meter curl provides (see the Progress meter section) works fine for uploads as well. What needs to be remembered is that the progress meter is automatically disabled when you are sending output to stdout, and most protocols curl support can output something even for an upload.

Therefore, you may need to explicitly redirect the downloaded data to a file (using shell redirect '>', `-o` or similar) to get the progress meter displayed for upload.

Rate limiting

Rate limiting works exactly the same for uploads as for downloads and curl, in fact, only has a single limit that will limit the speed in both directions.

See further details in the Download Rate limiting section.

Connections

Most of the protocols you use with curl speak TCP. With TCP, a client such as curl must first figure out the IP address(es) of the host you want to communicate with, then connect to it. "Connecting to it" means performing a TCP protocol handshake.

For ordinary command line usage, operating on a URL, these are details which are taken care of under the hood, and which you can mostly ignore. But at times you might find yourself wanting to tweak the specifics…

Name resolve tricks

Edit the hosts file

Maybe you want the command `curl http://example.com` to connect to your local server instead of the actual server.

You can normally and easily do that by editing your `hosts` file (`/etc/hosts` on Linux and Unix systems) and adding, for example, `127.0.0.1 example.com` to redirect the host to your localhost. However this edit requires admin access and it has the downside that it affects all other applications at the same time.

Change the Host: header

The `Host:` header is the normal way an HTTP client tells the HTTP server which server it speaks to, as typically an HTTP server serves many different names using the same software instance.

So, by passing in a custom modified `Host:` header you can have the server respond with the contents of the site even when you didn't actually connect to that host name.

For example, you run a test instance of your main site `www.example.com` on your local machine and you want to have curl ask for the index html:

```
curl -H "Host: www.example.com" http://localhost/
```

When setting a custom `Host:` header and using cookies, curl will extract the custom name and use that as host when matching cookies to send off.

The `Host:` header is not enough when communicating with an HTTPS server. With HTTPS there's a separate extension field in the TLS protocol called SNI (Server Name Indication) that lets the client tell the server the name of the server it wants to talk to. curl will only extract the SNI name to send from the given URL.

Provide a custom IP address for a name

Do you know better than the name resolver where curl should go? Then you can give an IP address to curl yourself! If you want to redirect port 80 access for `example.com` to instead reach your localhost:

```
curl --resolve example.com:80:127.0.0.1 http://example.com/
```

You can even specify multiple `--resolve` switches to provide multiple redirects of this sort, which can be handy if the URL you work with uses HTTP redirects or if you just want to have your command line work with multiple URLs.

`--resolve` inserts the address into curl's DNS cache, so it will effectively make curl believe that's the address it got when it resolved the name.

When talking HTTPS, this will send SNI for the name in the URL and curl will verify the server's response to make sure it serves for the name in the URL.

Provide a replacement name

As a close relative to the `--resolve` option, the `--connect-to` option provides a minor variation. It allows you to specify a replacement name and port number for curl to use under the hood when a specific name and port number is used to connect.

For example, suppose you have a single site called `www.example.com` that in turn is actually served by three different individual HTTP servers: load1, load2 and load3, for load balancing purposes. In a typical normal procedure, curl resolves the main site and gets to speak to one of the load balanced servers (as it gets a list back and just picks one of them) and all is well. If you want to send a test request to one specific server out of the load balanced set (`load1.example.com` for example) you can instruct curl to do that.

You *can* still use `--resolve` to accomplish this if you know the specific IP address of load1. But without having to first resolve and fix the IP address separately, you can tell curl:

```
curl --connect-to www.example.com:80:load1.example.com:80 http://www.example.com
```

It redirects from a SOURCE NAME + SOURCE PORT to a DESTINATION NAME + DESTINATION PORT. curl will then resolve the `load1.example.com` name and connect, but in all other ways still assume it is talking to `www.example.com`.

Name resolve tricks with c-ares

As should be detailed elsewhere in this book, curl may be built with several different name resolving backends. One of those backends is powered by the c-ares library and when curl is built to use c-ares, it gets a few extra superpowers that curl built to use other name resolve backends don't get. Namely, it gains the ability to more specifically instruct what DNS servers to use and how that DNS traffic is using the network.

With `--dns-servers`, you can specify exactly which DNS server curl should use instead of the default one. This lets you run your own experimental server that answers differently, or use a backup one if your regular one is unreliable or dead.

With `--dns-ipv4-addr` and `--dns-ipv6-addr` you ask curl to "bind" its local end of the DNS communication to a specific IP address and with `--dns-interface` you can instruct curl to use a specific network interface to use for its DNS requests.

These `--dns-*` options are very advanced and are only meant for people who know what they are doing and understand what these options do. But they offer very customizable DNS name resolution operations.

Connection timeout

curl will typically make a TCP connection to the host as an initial part of its network transfer. This TCP connection can fail or be very slow, if there are shaky network conditions or faulty remote servers.

To reduce the impact on your scripts or other use, you can set the maximum time in seconds which curl will allow for the connection attempt. With `--connect-timeout` you tell curl the maximum time to allow for connecting, and if curl has not connected in that time it returns a failure.

The connection timeout only limits the time curl is allowed to spend up until the moment it connects, so once the TCP connection has been established it can take longer time. See the Timeouts section for more on generic curl timeouts.

If you specify a low timeout, you effectively disable curl's ability to connect to remote servers, slow servers or servers you access over unreliable networks.

The connection timeout can be specified as a decimal value for sub-second precision. For example, to allow 2781 milliseconds to connect:

```
curl --connect-timeout 2.781 https://example.com/
```

Network interface

On machines with multiple network interfaces that are connected to multiple networks, there are situations where you can decide which network interface you would prefer the outgoing network traffic to use. Or which originating IP address (out of the multiple ones you have) to use in the communication.

Tell curl which network interface, which IP address or even host name that you would like to "bind" your local end of the communication to, with the `--interface` option:

```
curl --interface eth1 https://www.example.com/

curl --interface 192.168.0.2 https://www.example.com/

curl --interface machine2 https://www.example.com/
```

Local port number

A TCP connection is created between an IP address and a port number in the local end and an IP address and a port number in the remote end. The remote port number can be specified in the URL and usually helps identify which service you are targeting.

The local port number is usually randomly assigned to your TCP connection by the network stack and you normally don't have to think about it much further. However, in some circumstances you find yourself behind network equipment, firewalls or similar setups that put restrictions on what source port numbers that can be allowed to set up the outgoing connections.

For situations like this, you can specify which local ports curl should bind the the connection to. You can specify a single port number to use, or a range of ports. We recommend using a range because ports are scarce resources and the exact one you want may already be in use. If you ask for a local port number (or range) that curl can't obtain for you, it will exit with a failure.

Also, on most operating systems you cannot bind to port numbers below 1024 without having a higher privilege level (root) and we generally advise against running curl as root if you can avoid it.

Ask curl to use a local port number between 4000 and 4200 when getting this HTTPS page:

```
curl --local-port 4000-4200 https://example.com/
```

Keep alive

TCP connections can be totally without traffic in either direction when they are not used. A totally idle connection can therefore not be clearly separated from a connection that has gone completely stale because of network or server issues.

At the same time, lots of network equipment such as firewalls or NATs are keeping track of TCP connections these days, so that they can translate addresses, block "wrong" incoming packets, etc. These devices often count completely idle connections as dead after N minutes, where N varies between device to device but at times is as short as 10 minutes or even less.

One way to help avoid a really slow connection (or an idle one) getting treated as dead and wrongly killed, is to make sure TCP keep alive is used. TCP keepalive is a feature in the TCP protocol that makes it send "ping frames" back and forth when it would otherwise be totally idle. It helps idle connections to detect breakage even when no traffic is moving over it, and helps intermediate systems not consider the connection dead.

curl uses TCP keepalive by default for the reasons mentioned here. But there might be times when you want to *disable* keepalive or you may want to change the interval between the TCP "pings" (curl defaults to 60 seconds). You can switch off keepalive with:

```
curl --no-keepalive https://example.com/
```

or change the interval to 5 minutes (300 seconds) with:

```
curl --keepalive-time 300 https://example.com/
```

Timeouts

Network operations are by their nature rather unreliable or perhaps fragile operations as they depend on a set of services and networks to be up and working for things to work. The availability of these services can come and go and the performance of them may also vary greatly from time to time.

The design of TCP even allows the network to get completely disconnected for an extended period of time without it necessarily getting noticed by the participants in the transfer.

The result of this is that sometimes Internet transfers take a very long time. Further, most operations in curl have no time-out by default!

Maximum time allowed to spend

Tell curl with `-m / --max-time` the maximum time, in seconds, that you allow the command line to spend before curl exits with a timeout error code (28). When the set time has elapsed, curl will exit no matter what is going on at that moment—including if it is transferring data. It really is the maximum time allowed.

The given maximum time can be specified with a decimal precision; `0.5` means 500 milliseconds and `2.37` equals 2370 milliseconds.

Example:

```
curl --max-time 5.5 https://example.com/
```

(Your locale may use another symbol than a dot for expressing numerical fractions.)

Never spend more than this to connect

`--connect-timeout` limits the time curl will spend trying to connect to the host. All the necessary steps done before the connection is considered complete have to be completed within the given time frame. Failing to connect within the given time will cause curl to exit with a timeout exit code (28).

The given maximum connect time can be specified with a decimal precision; `0.5` means 500 milliseconds and `2.37` equals 2370 milliseconds:

```
curl --connect-timeout 2.37 https://example.com/
```

Transfer speeds slower than this means exit

Having a fixed maximum time for a curl operation can be cumbersome, especially if you, for example, do scripted transfers and the file sizes and transfer times vary a lot. A fixed timeout value then needs to be set unnecessarily high to cover for worst cases.

As an alternative to a fixed time-out, you can tell curl to abandon the transfer if it gets below a certain speed and stays below that threshold for a specific period of time.

For example, if a transfer speed goes below 1000 bytes per second during 15 seconds, stop it:

```
curl --speed-time 15 --speed-limit 1000 https://example.com/
```

Keep connections alive

curl enables TCP keep-alive by default. TCP keep-alive is a feature that makes the TCP stack send a probe to the other side when there's no traffic, to make sure that it is still there and "alive". By using keep-alive, curl is much more likely to discover that the TCP connection is dead.

Use `--keepalive-time` to specify how often in full seconds you would like the probe to get sent to the peer. The default value is 60 seconds.

Sometimes this probing disturbs what you are doing and then you can easily disable it with `--no-keepalive`.

.netrc

Unix systems have for a very long time offered a way for users to store their user name and password for remote FTP servers. ftp clients have supported this for decades and this way allowed users to quickly login to known servers without manually having to reenter the credentials each time. The `.netrc` file is typically stored in a user's home directory. (On Windows, curl will look for it with the name `_netrc`).

This being a widespread and well used concept, curl also supports it—if you ask it to. curl does not, however, limit this feature to FTP, but can get credentials for machines for any protocol with this. See further below for how.

The .netrc file format

The .netrc file format is simple: you specify lines with a machine name and follow that with lines for the login and password that are associated with that machine.

machine name

Identifies a remote machine name. curl searches the .netrc file for a machine token that matches the remote machine specified in the URL. Once a match is made, the subsequent .netrc tokens are processed, stopping when the end of file is reached or another machine is encountered.

login name

The user name string for the remote machine.

password string

Supply a password. If this token is present, curl will supply the specified string if the remote server requires a password as part of the login process. Note that if this token is present in the .netrc file you really **should** make sure the file is not readable by anyone besides the user.

An example .netrc for the host example.com with a user named 'daniel', using the password 'qwerty' would look like:

```
machine example.com
login daniel
password qwerty
```

Enable netrc

`-n, --netrc` tells curl to look for and use the .netrc file.

`--netrc-file [file]` is similar to `--netrc`, except that you also provide the path to the actual file to use. This is useful when you want to provide the information in another directory or with another file name.

`--netrc-optional` is similar to `--netrc`, but this option makes the .netrc usage optional and not mandatory as the `--netrc` option.

Proxies

A proxy is a machine or software that does something on behalf of you, the client.

You can also see it as a middle man that sits between you and the server you want to work with, a middle man that you connect to instead of the actual remote server. You ask the proxy to perform your desired operation for you and then it will run off and do that and then return back the data to you.

There are several different types of proxies and we shall list and discuss them further down in this section.

Discover your proxy

Some networks are setup to require a proxy in order for you to reach the Internet or perhaps that special network you are interested in. The use of proxies are introduced on your network by the people and management that run your network for policy or technical reasons.

In the networking space there are a few methods for the automatic detection of proxies and how to connect to them, but none of those methods are truly universal and curl supports none of them. Furthermore, when you communicate to the outside world through a proxy that often means that you have to put a lot of trust on the proxy as it will be able to see and modify all the non-secure network traffic you send or get through it. That trust is not easy to assume automatically.

If you check your browser's network settings, sometimes under an advanced settings tab, you can learn what proxy or proxies your browser is configured to use. Chances are very big that you should use the same one or ones when you use curl.

TBD: screenshots of how to find proxy settings in Firefox and Chrome?

PAC

Some network environments provides several different proxies that should be used in different situations, and a very customizable way to handle that is supported by the browsers. This is called "proxy auto-config", or PAC.

A PAC file contains a JavaScript function that decides which proxy a given network connection (URL) should use, and even if it should not use a proxy at all. Browsers most typically read the PAC file off a URL on the local network.

Since curl has no JavaScript capabilities, curl doesn't support PAC files. If your browser and network use PAC files, the easiest route forward is usually to read the PAC file manually and figure out the proxy you need to specify to run curl successfully.

Captive portals

(these aren't proxies but in the way)

TBD

Proxy type

curl supports several different types of proxies.

The default proxy type is HTTP so if you specify a proxy host name (or IP address) without a scheme part (the part that is often written as "http://") curl goes with assuming it's an HTTP proxy.

curl also allows a number of different options to set the proxy type instead of using the scheme prefix. See the SOCKS section below.

HTTP

An HTTP proxy is a proxy that the client speaks HTTP with to get the transfer done. curl will, by default, assume that a host you point out with `-x` or `--proxy` is an HTTP proxy, and unless you also specify a port number it will default to port 3128 (and the reason for that particular port number is purely historical).

If you want to request the example.com web page using a proxy on 192.168.0.1 port 8080, a command line could look like:

```
curl -x 192.168.0.1:8080 http:/example.com/
```

Recall that the proxy receives your request, forwards it to the real server, then reads the response from the server and then hands that back to the client.

If you enable verbose mode with `-v` when talking to a proxy, you will see that curl connects to the proxy instead of the remote server, and you will see that it uses a slightly different request line.

HTTPS and proxy

HTTPS was designed to allow and provide secure and safe end-to-end privacy from the client to the server (and back). In order to provide that when speaking to an HTTP proxy, the HTTP protocol has a special request that curl uses to setup a tunnel through the proxy that it then can encrypt and verify. This HTTP method is known as `CONNECT`.

When the proxy tunnels encrypted data through to the remote server after a CONNECT method sets it up, the proxy cannot see nor modify the traffic without breaking the encryption:

```
curl -x proxy.example.com:80 https://example.com/
```

MITM-proxies

MITM means Man-In-The-Middle. MITM-proxies are usually deployed by companies in "enterprise environments" and elsewhere, where the owners of the network have a desire to investigate even TLS encrypted traffic.

To do this, they require users to install a custom "trust root" (CA cert) in the client, and then the proxy terminates all TLS traffic from the client, impersonates the remote server and acts like a proxy. The proxy then sends back a generated certificate signed by the custom CA. Such proxy setups usually transparently capture all traffic from clients to TCP port 443 on a remote machine. Running curl in such a network would also get its HTTPS traffic captured.

This practice, of course, allows the middle man to decrypt and snoop on all TLS traffic.

Non-HTTP protocols over an HTTP proxy

An "HTTP proxy" means the proxy itself speaks HTTP. HTTP proxies are primarily used to proxy HTTP but it is also fairly common that they support other protocols as well. In particular, FTP is fairly commonly supported.

When talking FTP "over" an HTTP proxy, it is usually done by more or less pretending the other protocol works like HTTP and asking the proxy to "get this URL" even if the URL isn't using HTTP. This distinction is important because it means that when sent over an HTTP proxy like this, curl doesn't really speak FTP even though given an FTP URL; thus FTP-specific features will not work:

```
curl -x http://proxy.example.com:80 ftp://ftp.example.com/file.txt
```

What you can do instead then, is to "tunnel through" the HTTP proxy!

HTTP proxy tunneling

Most HTTP proxies allow clients to "tunnel through" it to a server on the other side. That's exactly what's done every time you use HTTPS through the HTTP proxy.

You tunnel through an HTTP proxy with curl using `-p` or `--proxytunnel`.

When you do HTTPS through a proxy you normally connect through to the default HTTPS remote TCP port number 443, so therefore you will find that most HTTP proxies white list and allow connections only to hosts on that port number and perhaps a few others. Most proxies will deny clients from connecting to just any random port (for reasons only the proxy administrators know).

Still, assuming that the HTTP proxy allows it, you can ask it to tunnel through to a remote server on any port number so you can do other protocols "normally" even when tunneling. You can do FTP tunneling like this:

```
curl -p -x http://proxy.example.com:80 ftp://ftp.example.com/file.txt
```

You can tell curl to use HTTP/1.0 in its CONNECT request issued to the HTTP proxy by using `--proxy1.0 [proxy]` instead of `-x`.

SOCKS types

SOCKS is a protocol used for proxies and curl supports it. curl supports both SOCKS version 4 as well as version 5, and both versions come in two flavors.

You can select the specific SOCKS version to use by using the correct scheme part for the given proxy host with `-x`, or you can specify it with a separate option instead of `-x`.

SOCKS4 is for the version 4 and SOCKS4a is for the version 4 without resolving the host name locally:

```
curl -x socks4://proxy.example.com http://www.example.com/

curl --socks4 proxy.example.com http://www.example.com/
```

The SOCKS4a versions:

```
curl -x socks4a://proxy.example.com http://www.example.com/

curl --socks4a proxy.example.com http://www.example.com/
```

SOCKS5 is for the version 5 and SOCKS5-hostname is for the version 5 without resolving the host name locally:

```
curl -x socks5://proxy.example.com http://www.example.com/

curl --socks5 proxy.example.com http://www.example.com/
```

The SOCKS5-hostname versions. This sends the host name to the server so there's no name resolving done locally:

```
curl -x socks5h://proxy.example.com http://www.example.com/

curl --socks5-hostname proxy.example.com http://www.example.com/
```

Proxy authentication

HTTP proxies can require authentication, so curl then needs to provide the proper credentials to the proxy to be allowed to use it, and failing to do will only make the proxy return back HTTP responses using code 407.

Authentication for proxies is very similar to "normal" HTTP authentication, but is separate from the server authentication to allow clients to independently use both normal host authentication as well as proxy authentication.

With curl, you set the user name and password for the proxy authentication with the `-U user:password` or `--proxy-user user:password` option:

```
curl -U daniel:secr3t -x myproxy:80 http://example.com
```

This example will default to using the Basic authentication scheme. Some proxies will require another authentication scheme (and the headers that are returned when you get a 407 response will tell you which) and then you can ask for a specific method with `--proxy-digest`, `--proxy-negotiate`, `--proxy-ntlm`. The above example command again, but asking for NTLM auth with the proxy:

```
curl -U daniel:secr3t -x myproxy:80 http://example.com --proxy-ntlm
```

There's also the option that asks curl to figure out which method the proxy wants and supports and then go with that (with the possible expense of extra roundtrips) using `--proxy-anyauth`. Asking curl to use any method the proxy wants is then like this:

```
curl -U daniel:secr3t -x myproxy:80 http://example.com --proxy-anyauth
```

HTTPS to proxy

All the previously mentioned protocols to speak with the proxy are clear text protocols, HTTP and the SOCKS versions. Using these methods could allow someone to eavesdrop on your traffic the local network where you or the proxy reside.

One solution for that is to use HTTPS to the proxy, which then establishes a secure and encrypted connection that is safe from easy surveillance.

Proxy environment variables

curl checks for the existence of specially-named environment variables before it runs to see if a proxy is requested to get used.

You specify the proxy by setting a variable named `[scheme]_proxy` to hold the proxy host name (the same way you would specify the host with `-x`). So if you want to tell curl to use a proxy when access a HTTP server, you set the 'http_proxy' environment variable. Like this:

```
http_proxy=http://proxy.example.com:80
curl -v www.example.com
```

While the above example shows HTTP, you can, of course, also set ftp_proxy, https_proxy, and so on. All these proxy environment variable names except http_proxy can also be specified in uppercase, like HTTPS_PROXY.

To set a single variable that controls *all* protocols, the ALL_PROXY exists. If a specific protocol variable one exists, such a one will take precedence.

When using environment variables to set a proxy, you could easily end up in a situation where one or a few host names should be excluded from going through the proxy. This is then done with the NO_PROXY variable. Set that to a comma- separated list of host names that should not use a proxy when being accessed. You can set NO_PROXY to be a single asterisk ('*') to match all hosts.

As an alternative to the NO_PROXY variable, there's also a `--noproxy` command line option that serves the same purpose and works the same way.

Proxy headers

--proxy-header

TBD

Exit status

A lot of effort has gone into the project to make curl return a usable exit code when something goes wrong and it will always return 0 (zero) when the operation went as planned.

If you write a shell script or batch file that invokes curl, you can always check the return code to detect problems in the invoked command. Below, you will find a list of return codes as of the time of this writing. Over time we tend to slowly add new ones so if you get a code back not listed here, please refer to more updated curl documentation for aid.

A very basic Unix shell script could look like something like this:

```
#!/bin/sh
curl http://example.com
res=$?
if test "$res" != "0"; then
   echo "the curl command failed with: $res"
fi
```

Available exit codes

1. Unsupported protocol. This build of curl has no support for this protocol. Usually this happens because the URL was misspelled to use a scheme part that either has a space in front of it or spells "http" like "htpt" or similar. Another common mistake is that you use a libcurl installation that was built with one or more protocols disabled and you now ask libcurl to use one of those protocols that were disabled in the build.

2. Failed to initialize. This is mostly an internal error or a problem with the libcurl installation or system libcurl runs in.

3. URL malformed. The syntax was not correct. This happens when you mistype a URL so that it ends up wrong, or in rare situations you are using a URL that is accepted by another tool that curl doesn't support only because there is no universal URL standard that everyone adheres to.

4. A feature or option that was needed to perform the desired request was not enabled or was explicitly disabled at build-time. To make curl able to do this, you probably need another build of libcurl!

5. Couldn't resolve proxy. The address of the given proxy host could not be resolved. Either the given proxy name is just wrong, or the DNS server is misbehaving and doesn't know about this name when it should or perhaps even the system you run curl on is misconfigured so that it doesn't find/use the correct DNS server.
6. Couldn't resolve host. The given remote host's address was not resolved. The address of the given server could not be resolved. Either the given host name is just wrong, or the DNS server is misbehaving and doesn't know about this name when it should or perhaps even the system you run curl on is misconfigured so that it doesn't find/use the correct DNS server.
7. Failed to connect to host. curl managed to get an IP address to the machine and it tried to setup a TCP connection to the host but failed. This can be because you have specified the wrong port number, entered the wrong host name, the wrong protocol or perhaps because there is a firewall or another network equipment in between that blocks the traffic from getting through.
8. Unknown FTP server response. The server sent data curl couldn't parse. This is either because of a bug in curl, a bug in the server or because the server is using an FTP protocol extension that curl doesn't support. The only real work-around for this is to tweak curl options to try it to use other FTP commands that perhaps won't get this unknown server response back.
9. FTP access denied. The server denied login or denied access to the particular resource or directory you wanted to reach. Most often you tried to change to a directory that doesn't exist on the server. The directory of course is what you specify in the URL.
10. FTP accept failed. While waiting for the server to connect back when an active FTP session is used, an error code was sent over the control connection or similar.
11. FTP weird PASS reply. Curl couldn't parse the reply sent to the PASS request. PASS in the command curl sends the password to the server with, and even anonymous connections to FTP server actually sends a password - a fixed anonymous string. Getting a response back from this command that curl doesn't understand is a strong indication that this isn't an FTP server at all or that the server is badly broken.
12. During an active FTP session (PORT is used) while waiting for the server to connect, the timeout expired. It took too long for the server to get back. This is usually a sign that something is preventing the server from reaching curl successfully. Like a firewall or other network arrangements. .
13. Unknown response to FTP PASV command, Curl couldn't parse the reply sent to the PASV request. This is a strange server. PASV is used to setup the second data transfer connection in passive mode, see the FTP uses two connections section for more on

that. You might be able to work-around this problem by using PORT instead, with the `--ftp-port` option.

14. Unknown FTP 227 format. Curl couldn't parse the 227-line the server sent. This is most certainly a broken server. A 227 is the FTP server's response when sending back information on how curl should connect back to it in passive mode. You might be able to work-around this problem by using PORT instead, with the `--ftp-port` option.
15. FTP can't get host. Couldn't use the host IP address we got in the 227-line. This is most likely an internal error!
16. HTTP/2 error. A problem was detected in the HTTP2 framing layer. This is somewhat generic and can be one out of several problems, see the error message for details.
17. FTP couldn't set binary. Couldn't change transfer method to binary. This server is broken. curl needs to set the transfer to the correct mode before it is started as otherwise the transfer can't work.
18. Partial file. Only a part of the file was transferred. When the transfer is considered complete, curl will verify that it actually received the same amount of data that it was told before-hand that it was going to get. If the two numbers don't match, this is the error code. It could mean that curl got fewer bytes than advertised or that it got more. curl itself cannot know which number that is wrong or which is correct. If any.
19. FTP couldn't download/access the given file. The RETR (or similar) command failed. curl got an error from the server when trying to download the file.
20. **Not used**
21. Quote error. A quote command returned an error from the server. curl allows several different ways to send custom commands to a IMAP, POP3, SMTP or FTP server and features a generic check that the commands work. When any of the individually issued commands fails, this is exit status is returned. The advice is generally to watch the headers in the FTP communication to better understand exactly what failed and how.
22. HTTP page not retrieved. The requested url was not found or returned another error with the HTTP error code being 400 or above. This return code only appears if `-f, --fail` is used.
23. Write error. Curl couldn't write data to a local filesystem or similar. curl receives data chunk by chunk from the network and it stores it like at (or writes it to stdout), one piece at a time. If that write action gets an error, this is the exit status.
24. **Not used**

25. Upload failed. The server refused to accept or store the file that curl tried to send to it. This is usually due to wrong access rights on the server but can also happen due to out of disk space or other resource constraints. This error can happen for many protocols.
26. Read error. Various reading problems. The inverse to exit status 23. When curl sends data to a server, it reads data chunk by chunk from a local file or stdin or similar, and if that reading fails in some way this is the exit status curl will return.
27. Out of memory. A memory allocation request failed. curl needed to allocate more memory than what the system was willing to give it and curl had to exit. Try using smaller files or make sure that curl gets more memory to work with.
28. Operation timeout. The specified time-out period was reached according to the conditions. curl offers several timeouts, and this exit code tells one of those timeout limits were reached. Extend the timeout or try changing something else that allows curl to finish its operation faster. Often, this happens due to network and remote server situations that you cannot affect locally.
29. **Not used**
30. FTP PORT failed. The PORT command failed. Not all FTP servers support the PORT command; try doing a transfer using PASV instead! The PORT command is used to ask the server to create the data connection by *connecting back* to curl. See also the FTP uses two connections section.
31. FTP couldn't use REST. The REST command failed. This command is used for resumed FTP transfers. curl needs to issue the REST command to do range or resumed transfers. The server is broken, try the same operation without range/resume as a crude work-around!
32. **Not used**
33. HTTP range error. The range request didn't work. Resumed HTTP requests aren't necessary acknowledged or supported, so this exit code signals that for this resource on this server, there can be no range or resumed transfers.
34. HTTP post error. Internal post-request generation error. If you get this error, please report the exact circumstances to the curl project!
35. A TLS/SSL connect error. The SSL handshake failed. The SSL handshake can fail due to numerous different reasons so the error message may offer some additional clues. Maybe the parties couldn't agree to a SSL/TLS version, an agreeable cipher suite or similar.

36. Bad download resume. Couldn't continue an earlier aborted download. When asking to resume a transfer that then ends up not possible to do, this error can get returned. For FILE, FTP or SFTP.
37. Couldn't read the given file when using the FILE:// scheme. Failed to open the file. The file could be non-existing or is it a permission problem perhaps?
38. LDAP cannot bind. LDAP "bind" operation failed, which is a necessary step in the LDAP operation and thus this means the LDAP query could not be performed. This might happen because of wrong username or password, or for other reasons.
39. LDAP search failed. The given search terms caused the LDAP search to return an error.
40. **Not used**
41. **Not used**
42. Aborted by callback. An application told libcurl to abort the operation. This error code is not generally made visible to users and not to users of the curl tool.
43. Bad function argument. A function was called with a bad parameter - this return code is present to help application authors to understand why libcurl can't perform certain actions and should never be return by the curl tool. Please file a bug report to the curl project if this happens to you!
44. **Not used**
45. Interface error. A specified outgoing network interface could not be used. curl will typically decide outgoing network and IP addresses by itself but when explicitly asked to use a specific one that curl can't use, this error can occur.
46. **Not used**
47. Too many redirects. When following HTTP redirects, libcurl hit the maximum number set by the application. The maximum number of redirects is unlimited by libcurl but is set to 50 by default by the curl tool. The limit is present to stop endless redirect loops. Change the limit with `--max-redirs`.
48. Unknown option specified to libcurl. This could happen if you use a curl version that is out of sync with the underlying libcurl version. Perhaps your newer curl tries to use an option in the older libcurl that wasn't introduced until after the libcurl version you're using but is known to your curl tool code as that is newer. To decrease the risk of this and make sure it doesn't happen: use curl and libcurl of the same version number.
49. Malformed telnet option. The telnet options you provide to curl was not using the correct syntax.

50. **Not used**
51. The server's SSL/TLS certificate or SSH fingerprint failed verification. curl can then not be sure of the server being who it claims to be. See the using TLS with curl section for more TLS details and using SCP and SFTP with curl for more SSH specific details.
52. The server didn't reply anything, which in this context is considered an error. When a HTTP(S) server responds to a HTTP(S) request, it will always return *something* as long as it is alive and sound. All valid HTTP responses have a status line and responses header. Not getting anything at all back is an indication the server is faulty or perhaps that something prevented curl from reaching the right server or that you're trying to connect to the wrong port number etc.
53. SSL crypto engine not found.
54. Cannot set SSL crypto engine as default.
55. Failed sending network data. Sending data over the network is a crucial part of most curl operations and when curl gets an error from the lowest networking layers that the sending failed, this exit status gets returned. To pinpoint why this happens, some serious digging is usually required. Start with enabling verbose mode, do tracing and if possible check the network traffic with a tool like Wireshark or similar.
56. Failure in receiving network data. Receiving data over the network is a crucial part of most curl operations and when curl gets an error from the lowest networking layers that the receiving of data failed, this exit status gets returned. To pinpoint why this happens, some serious digging is usually required. Start with enabling verbose mode, do tracing and if possible check the network traffic with a tool like Wireshark or similar.
57. **Not used**
58. Problem with the local certificate. The client certificate had a problem so it couldn't be used. Permissions? The wrong pass phrase?
59. Couldn't use the specified SSL cipher.
60. Peer certificate cannot be authenticated with known CA certificates.
61. Unrecognized transfer encoding.
62. Invalid LDAP URL.
63. Maximum file size exceeded.
64. Requested FTP SSL level failed.
65. Sending the data requires a rewind that failed.

66. Failed to initialize SSL Engine.
67. The user name, password, or similar was not accepted and curl failed to log in.
68. File not found on TFTP server.
69. Permission problem on TFTP server.
70. Out of disk space on TFTP server.
71. Illegal TFTP operation.
72. Unknown TFTP transfer ID.
73. File already exists (TFTP).
74. No such user (TFTP).
75. Character conversion failed.
76. Character conversion functions required.
77. Problem with reading the SSL CA cert
78. The resource referenced in the URL does not exist.
79. An unspecified error occurred during the SSH session.
80. Failed to shut down the SSL connection.
81. **Not used**
82. Could not load CRL file, missing or wrong format
83. TLS certificate issuer check failed
84. The FTP PRET command failed
85. RTSP: mismatch of CSeq numbers
86. RTSP: mismatch of Session Identifiers
87. unable to parse FTP file list
88. FTP chunk callback reported error
89. No connection available, the session will be queued
90. SSL public key does not matched pinned public key
91. Invalid SSL certificate status
92. Stream error in HTTP/2 framing layer

Error message

When curl exits with a non-zero code, it will also output an error message (unless `--silent` is used). That error message may add some additional information or circumstances to the exit status number itself so the same error number can get different error messages.

"Not used"

The list of exit codes above contains a number of values marked as 'not used'. Those are exit status codes that aren't used in modern versions of curl but that have been used or were intended to be used in the past. They may very well be used in a future version of curl.

Additionally, the highest used error status in this list is 92, but there is no guarantee that a future curl version won't decide to add more exit codes after that number.

FTP

FTP, the File Transfer Protocol, is probably the oldest network protocol that curl supports—it was created in the early 1970s. The official spec that still is the go-to documentation is RFC 959, from 1985, published well over a decade before the first curl release.

FTP was created in a different era of the Internet and computers and as such it works a little bit differently than most other protocols. These differences can often be ignored and things will just work, but they are also important to know at times when things don't run as planned.

Ping-pong

The FTP protocol is a command and response protocol; the client sends a command and the server responds. If you use curl's `-v` option you will get to see all the commands and responses during a transfer.

For an ordinary transfer, there are something like 5 to 8 commands necessary to send and as many responses to wait for and read. Perhaps needlessly to say, if the server is in a remote location there will be a lot of time waiting for the ping pong to go through before the actual file transfer can be set up and get started. For small files, the initial commands can very well take longer time than the actual data transfer.

Transfer mode

When an FTP client is about to transfer data, it specifies to the server which "transfer mode" it would like the upcoming transfer to use. The two transfer modes curl supports are 'ASCII' and 'BINARY'. Ascii is basically for text and usually means that the server will send the files with converted newlines while binary means sending the data unaltered and assuming the file is not text.

curl will default to binary transfer mode for FTP, and you ask for ascii mode instead with `-B, --use-ascii` or by making sure the URL ends with `;type=A`.

Authentication

FTP is one of the protocols you normally don't access without a user name and password. It just happens that for systems that allow "anonymous" FTP access you can login with pretty much any name and password you like. When curl is used on an FTP URL to do transfer without any given user name or password, it uses the name `anonymous` with the password `ftp@example.com`.

If you want to provide another user name and password, you can pass them on to curl either with the `-u, --user` option or embed the info in the URL:

curl --user daniel:secret ftp://example.com/download

curl ftp://daniel:secret@example.com/download

FTP uses two connections

FTP uses two TCP connections! The first connection is setup by the client when it connects to an FTP server, and is called the *control connection*. As the initial connection, it gets to handle authentication and changing to the correct directory on the remote server, etc. When the client then is ready to transfer a file, a second TCP connection is established and the data is transferred over that.

This setting up of a second connection causes nuisances and headaches for several reasons.

Active connections

The client can opt to ask the server to connect to the client to set it up, a so-called "active" connection. This is done with the PORT or EPRT commands. Allowing a remote host to connect back to a client on a port that the client opens up requires that there's no firewall or other network appliance in between that refuses that to go through and that is far from always the case. You ask for an active transfer using `curl -P [arg]` (also known as `--ftp-port` in long form) and while the option allows you to specify exactly which address to use, just setting the same as you come from is almost always the correct choice and you do that with `-P -`, like this way to ask for a file:

```
curl -P - ftp://example.com/foobar.txt
```

You can also explicitly ask curl to not use EPRT (which is a slightly newer command than PORT) with the `--no-eprt` command-line option.

Passive connections

Curl defaults to asking for a "passive" connection, which means it sends a PASV or EPSV command to the server and then the server opens up a new port for the second connection that then curl connects to. Outgoing connections to a new port are generally easier and less restricted for end users and clients, but it then requires that the network in the server's end allows it.

Passive connections are enabled by default, but if you have switched on active before, you can switch back to passive with `--ftp-pasv`.

You can also explicitly ask curl not to use EPSV (which is a slightly newer command than PASV) with the `--no-epsv` command-line option.

Sometimes the server is running a funky setup so that when curl issues the PASV command and the server responds with an IP address for curl to connect to, that address is wrong and then curl fails to setup the data connection. For this (hopefully rare) situation, you can ask curl to ignore the IP address mentioned in the PASV response (`--ftp-skip-pasv-ip`) and instead use the same IP address it has for the control connection even for the second connection.

Firewall issues

Using either active or passive transfers, any existing firewalls in the network path pretty much have to have stateful inspection of the FTP traffic to figure out the new port to open that up and accept it for the second connection.

How to traverse directories

When doing FTP commands to traverse the remote file system, there are a few different ways curl can proceed to reach the target file, the file the user wants to transfer.

multicwd

curl can do one change directory (CWD) command for every individual directory down the file tree hierarchy. If the full path is `one/two/three/file.txt`, that method means doing three `CWD` commands before asking for the `file.txt` file to get transferred. This method thus creates quite a large number of commands if the path is many levels deep. This method is mandated by an early spec (RFC 1738) and is how curl acts by default:

```
curl --ftp-method multicwd ftp://example.com/one/two/three/file.txt
```

This then equals this FTP command/response sequence (simplified):

```
> CWD one
< 250 OK. Current directory is /one
> CWD two
< 250 OK. Current directory is /one/two
> CWD three
< 250 OK. Current directory is /one/two/three
> RETR file.txt
```

nocwd

The opposite to doing one CWD for each directory part is to not change the directory at all. This method asks the server using the entire path at once and is thus very fast. Occasionally servers have a problem with this and it isn't purely standards-compliant:

```
curl --ftp-method nocwd ftp://example.com/one/two/three/file.txt
```

This then equals this FTP command/response sequence (simplified):

```
> RETR one/two/three/file.txt
```

singlecwd

This is the in-between the other two FTP methods. This makes a single `CWD` command to the target directory and then it asks for the given file:

```
curl --ftp-method singlecwd ftp://example.com/one/two/three/file.txt
```

This then equals this FTP command/response sequence (simplified):

```
> CWD one/two/three
< 250 OK. Current directory is /one/two/three
> RETR file.txt
```

More advanced FTP

FTP Directory listing

You can list a remote FTP directory with curl by making sure the URL ends with a trailing slash. If the URL ends with a slash, curl will presume that it is a directory you want to list. If it isn't actually a directory, you will most likely instead get an error.

```
curl ftp://ftp.example.com/directory/
```

With FTP there is no standard syntax for the directory output that is returned for this sort of command that uses the standard FTP command `LIST`. The listing is usually humanly readable and perfectly understandable but you will see that different servers will return the listing in slightly different ways.

One way to get just a listing of all the names in a directory and thus to avoid the special formatting of the regular directory listings is to tell curl to `--list-only` (or just `-l`). curl then issues the `NLST` FTP command instead:

```
curl --list-only ftp://ftp.example.com/directory/
```

NLST has its own quirks though, as some FTP servers list only actual *files* in their response to NLST; they do not include directories and symbolic links!

Uploading with FTP

To upload to an FTP server, you specify the entire target file path and name in the URL, and you specify the local file name to upload with `-T, --upload-file`. Optionally, you end the target URL with a slash and then the file component from the local path will be appended by curl and used as the remote file name.

Like:

```
curl -T localfile ftp://ftp.example.com/dir/path/remote-file
```

or to use the local file name as the remote name:

```
curl -T localfile ftp://ftp.example.com/dir/path/
```

curl also supports globbing in the -T argument so you can opt to easily upload a range or a series of files:

```
curl -T image[1-99].jpg ftp://ftp.example.com/upload/
```

or

```
curl -T '{Huey,Dewey,Louie}.jpg' ftp://ftp.example.com/nephews/
```

Custom FTP commands

TBD

FTPS

TBD

Common FTP problems

TBD

SCP and SFTP

curl supports the SCP and SFTP protocols if built with the correct prerequisite 3rd party library, libssh2.

SCP and SFTP are both protocols that are built on top of SSH, a secure and encrypted data protocol that is similar to TLS but differs in a few important ways. For example, SSH doesn't use certificates of any sort but instead it uses public and private keys. Both SSH and TLS provide strong crypto and secure transfers when used correctly.

The SCP protocol is generally considered to be the black sheep of the two since it isn't very portable and usually only works between Unix systems.

URLs

SFTP and SCP URLs are similar to other URLs and you download files using these protocols the same as with others:

```
curl sftp://example.com/file.zip -u user
```

and:

```
curl scp://example.com/file.zip -u user
```

SFTP (but not SCP) supports getting a file listing back when the URL ends with a trailing slash:

```
curl sftp://example.com/ -u user
```

Note that both these protocols work with "users" and you don't ask for a file anonymously or with a standard generic name. Most systems will require that users authenticate, as outlined below.

When requesting a file from an SFTP or SCP URL, the file path given is considered to be the absolute path on the remote server unless you specifically ask for the path relative to the user's home directory. You do that by making sure the path starts with `/~/` . This is quite the opposite to how FTP URLs work and is a common cause for confusion among users.

For user `daniel` to transfer `todo.txt` from his home directory, it would look similar to this:

```
curl sftp://example.com/~/todo.txt -u daniel
```

Auth

TBD

Known hosts

A secure network client needs to make sure that the remote host is exactly the host that it thinks it is communicating with. With TLS based protocols, it is done by the client verifying the server's certificate.

With SSH protocols there are no server certificates, but instead each server can provide its unique key. And unlike TLS, SSH has no certificate authorities or anything so the client simply has to make sure that the host's key matches what it already knows (via other means) it should look like.

The matching of keys is typically done using hashes of the key and the file that the client stores the hashes for known servers is often called `known_hosts` and is put in a dedicated SSH directory. On Linux systems that is usually called `~/.ssh` .

When curl connects to a SFTP and SCP host, it will make sure that the host's key hash is already present in the known hosts file or it will deny continued operation because it cannot trust that the server is the right one. Once the correct hash exists in `known_hosts` curl can perform transfers.

To force curl to skip checking and obeying to the the `known_hosts` file, you can use the `-k / --insecure` command-line option. You must use this option with extreme care since it makes it possible for man-in-the-middle attacks not to be detected.

Reading email

There are two dominant protocols on the Internet for reading/downloading email from servers (at least if we don't count web based reading), and they are IMAP and POP3. The former being the slightly more modern alternative. curl supports both.

POP3

To list message numbers and sizes:

```
curl pop3://mail.example.com/
```

To download message 1:

```
curl pop3://mail.example.com/1
```

To delete message 1:

```
curl --request DELE pop3://mail.example.com/1
```

IMAP

TBD

SMTP

SMTP stands for Simple Mail Transfer Protocol.

curl supports sending data to a an SMTP server, which combined with the right set of command line options makes an email get sent to a set of receivers of your choice.

When sending SMTP with curl, there are a two necessary command line options that **must** be used.

- You need to tell the server at least one recipient with `--mail-rcpt`. You can use this option several times and then curl will tell the server that all those email addresses should receive the email.
- You need to tell the server which email address that is the sender of the email with `--mail-from`. It is important to realize that this email address is not necessarily the same as is shown in the `From:` line of the email text.

Then, you need to provide the actual email data. This is a (text) file formatted according to RFC 5322. It is a set of headers and a body. Both the headers and the body need to be correctly encoded. The headers typically include `To:`, `From:`, `Subject:`, `Date:` etc.

A basic command to send an email:

```
curl smtp://mail.example.com --mail-from myself@example.com --mail-rcpt
receiver@example.com --upload-file email.txt
```

Example email.txt

```
From: John Smith <john@example.com>
To: Joe Smith <smith@example.com>
Subject: an example.com example email
Date: Mon, 7 Nov 2016 08:45:16

Dear Joe,
Welcome to this example email. What a lovely day.
```

Secure mail transfer

Some mail providers allow or require using SSL for SMTP. They may use a dedicated port for SSL or allow SSL upgrading over a plaintext connection.

If your mail provider has a dedicated SSL port you can use smtps:// instead of smtp://, which uses the SMTP SSL port of 465 by default and requires the entire connection to be SSL. For example smtps://smtp.gmail.com/.

However, if your provider allows upgrading from plaintext to secure transfers you can use one of these options:

```
--ssl           Try SSL/TLS (FTP, IMAP, POP3, SMTP)
--ssl-reqd      Require SSL/TLS (FTP, IMAP, POP3, SMTP)
```

You can tell curl to *try* but not require upgrading to secure transfers by adding `--ssl` to the command:

```
curl --ssl smtp://mail.example.com --mail-from myself@example.com
     --mail-rcpt receiver@example.com --upload-file email.txt
```

You can tell curl to *require* upgrading to using secure transfers by adding `--ssl-reqd` to the command:

```
curl --ssl-reqd smtp://mail.example.com --mail-from myself@example.com
     --mail-rcpt receiver@example.com --upload-file email.txt
```

The SMTP URL

The path part of a SMTP request specifies the host name to present during communication with the mail server. If the path is omitted then curl will attempt to figure out the local computer's host name and use that. However, this may not return the fully qualified domain name that is required by some mail servers and specifying this path allows you to set an alternative name, such as your machine's fully qualified domain name, which you might have obtained from an external function such as gethostname or getaddrinfo.

To connect to the mail server at `mail.example.com` and send your local computer's host name in the HELO / EHLO command:

```
curl smtp://mail.example.com
```

You can of course as always use the `-v` option to get to see the client-server communication.

To instead have curl send `client.example.com` in the `HELO` / `EHLO` command to the mail server at `mail.example.com` , use:

```
curl smtp://mail.example.com/client.example.com
```

No MX lookup!

When you send email with an ordinary mail client, it will first check for an MX record for the particular domain you want to send email to. If you send an email to joe@example.com, the client will get the MX records for `example.com` to learn which mail server(s) to use when sending email to example.com users.

curl does no MX lookups by itself. If you want to figure out which server to send an email to for a particular domain, we recommend you figure that out first and then call curl to use those servers. Useful command line tools to get MX records with include 'dig' and 'nslookup'.

TELNET

TBD

TLS

TLS stands for Transport Layer Security and is the name for the technology that was formerly called SSL. The term SSL hasn't really died though so these days both the terms TLS and SSL are often used interchangeably to describe the same thing.

TLS is a cryptographic security layer "on top" of TCP that makes the data tamper proof and guarantees server authenticity, based on strong public key cryptography and digital signatures.

Ciphers

When curl connections to a TLS server, it negotiates how to speak the protocol and that negotiation involves several parameters and variables that both parties need to agree to. One of the parameters is which cryptography algorithms to use, the so called cipher. Over time, security researchers figure out flaws and weaknesses in existing ciphers and they are gradually phased out over time.

Using the verbose option, `-v`, you can get information about which cipher and TLS version are negotiated. By using the `--ciphers` option, you can change what cipher to prefer in the negotiation, but mind you, this is a power feature that takes knowledge to know how to use in ways that don't just make things worse.

Enable TLS

curl supports the TLS version of many protocols. HTTP has HTTPS, FTP has FTPS, LDAP has LDAPS, POP3 has POP3S, IMAP has IMAPS and SMTP has SMTPS.

If the server side supports it, you can use the TLS version of these protocols with curl.

There are two general approaches to do TLS with protocols. One of them is to speak TLS already from the first connection handshake while the other is to "upgrade" the connection from plain-text to TLS using protocol specific instructions.

With curl, if you explicitly specify the TLS version of the protocol (the one that has a name that ends with an 'S' character) in the URL, curl will try to connect with TLS from start, while if you specify the non-TLS version in the URL you can *usually* upgrade the connection to TLS-based with the `--ssl` option.

The support table looks like this:

Clear	TLS version	--ssl
HTTP	HTTPS	no
LDAP	LDAPS	no
FTP	FTPS	**yes**
POP3	POP3S	**yes**
IMAP	IMAPS	**yes**
SMTP	SMTPS	**yes**

The protocols that *can* do `--ssl` all favor that method. Using `--ssl` means that curl will *attempt* to upgrade the connection to TLS but if that fails, it will still continue with the transfer using the plain-text version of the protocol. To make the `--ssl` option **require** TLS to continue, there's instead the `--ssl-reqd` option which will make the transfer fail if curl cannot successfully negotiate TLS.

Require TLS security for your FTP transfer:

```
curl --ssl-reqd ftp://ftp.example.com/file.txt
```

Suggest TLS to be used for your FTP transfer:

```
curl --ssl ftp://ftp.example.com/file.txt
```

Connecting directly with TLS (to HTTPS://, LDAPS://, FTPS:// etc) means that TLS is mandatory and curl will return an error if TLS isn't negotiated.

Get a file over HTTPS:

```
curl https://www.example.com/
```

SSL and TLS versions

SSL was invented in the mid 90s and has developed ever since. SSL version 2 was the first widespread version used on the Internet but that was deemed insecure already a very long time ago. SSL version 3 took over from there, and it too has been deemed not safe enough for use.

TLS version 1.0 was the first "standard". RFC 2246 was published 1999. After that, TLS 1.1 came and and in 2017, TLS 1.2 is still the gold standard. TLS 1.3 is in the works and we expect to see it finalized and published as a standard by the IETF at some point during

2018.

curl is designed to use a "safe version" of SSL/TLS by default. It means that it will not negotiate SSLv2 or SSLv3 unless specifically told to, and in fact several TLS libraries no longer provide support for those protocols so in many cases curl is not even able to speak those protocol versions unless you make a serious effort.

Option	Use
--sslv2	SSL version 2
--sslv3	SSL version 3
--tlsv1	TLS >= version 1.0
--tlsv1.0	TLS version 1.0
--tlsv1.1	TLS version 1.1
--tlsv1.2	TLS version 1.2
--tlsv1.3	TLS version 1.3

NOTE: TLS version 1.3 is only supported in selected very recent development versions of certain TLS libraries and requires curl 7.52.0 or later.

Verifying server certificates

Having a secure connection to a server is not worth a lot if you cannot also be certain that you are communicating with the **correct** host. If we don't know that, we could just as well be talking with an imposter that just *appears* to be who we think it is.

To check that it communicates with the right TLS server, curl uses a set of locally stored CA certificates to verify the signature of the server's certificate. All servers provide a certificate to the client as part of the TLS handshake and all public TLS-using servers have acquired that certificate from an established Certificate Authority.

After some applied crypto magic, curl knows that the server is in fact the correct one that acquired that certificate for the host name that curl used to connect to it. Failing to verify the server's certificate is a TLS handshake failure and curl exits with an error.

In rare circumstances, you may decide that you still want to communicate with a TLS server even if the certificate verification fails. You then accept the fact that your communication may be subject to Man-In-The-Middle attacks. You lower your guards with the `-k` or `--insecure` option.

CA store

curl needs a "CA store", a collection of CA certificates, to verify the TLS server it talks to.

If curl is built to use a TLS library that is "native" to your platform, chances are that library will use the native CA store as well. If not, curl has to either have been built to know where the local CA store is, or users need to provide a path to the CA store when curl is invoked.

You can point out a specific CA bundle to use in the TLS handshake with the `--cacert` command line option. That bundle needs to be in PEM format. You can also set the environment variable `CURL_CA_BUNDLE` to the full path.

CA store on windows

curl built on windows that isn't using the native TLS library (Schannel), have an extra sequence for how the CA store can be found and used.

curl will search for a CA cert file named "curl-ca-bundle.crt" in these directories and in this order:

1. application's directory
2. current working directory
3. Windows System directory (e.g. `C:\windows\system32`)
4. Windows Directory (e.g. `C:\windows`)
5. all directories along `%PATH%`

Certificate pinning

TLS certificate pinning is a way to verify that the public key used to sign the servers certificate has not changed. It is "pinned".

When negotiating a TLS or SSL connection, the server sends a certificate indicating its identity. A public key is extracted from this certificate and if it does not exactly match the public key provided to this option, curl will abort the connection before sending or receiving any data.

You tell curl a file name to read the sha256 value from, or you specify the base64 encoded hash directly in the command line with a "sha256//" prefix. You can specify one or more hashes like that, separated with semicolons (;).

```
curl --pinnedpubkey "sha256//83d34tasd3rt..." https://example.com/
```

This feature is not supported by all TLS backends.

OCSP stapling

This uses the TLS extension called Certificate Status Request to ask the server to provide a fresh "proof" from the CA in the handshake, that the certificate that it returns is still valid. This is a way to make really sure the server's certificate hasn't been revoked.

If the server doesn't support this extension, the test will fail and curl returns an error. And it is still far too common that servers don't support this.

Ask for the handshake to use the status request like this:

```
curl --cert-status https://example.com/
```

This feature is only supported by the OpenSSL, GnuTLS and NSS backends.

Client certificates

TLS client certificates are a way for clients to cryptographically prove to servers that they are truly the right peer. A command line that uses a client certificate specifies the certificate and the corresponding key, and they are then passed on the TLS handshake with the server.

You need to have your client certificate already stored in a file when doing this and you should supposedly have gotten it from the right instance via a different channel previously.

The key is typically protected by a password that you need to provide or get prompted for interactively.

curl offers options to let you specify a single file that is both the client certificate and the private key concatenated using `--cert`, or you can specify the key file independently with `--key`:

```
curl --cert mycert:mypassword https://example.com
curl --cert mycert:mypassword --key mykey https://example.com
```

For some TLS backends you can also pass in the key and certificate using different types:

```
curl --cert mycert:mypassword --cert-type PEM \
     --key mykey --key-type PEM https://example.com
```

TLS auth

TBD

Different TLS backends

TBD

Multiple TLS backends

TBD

When things don't run the way you thought they would

TBD

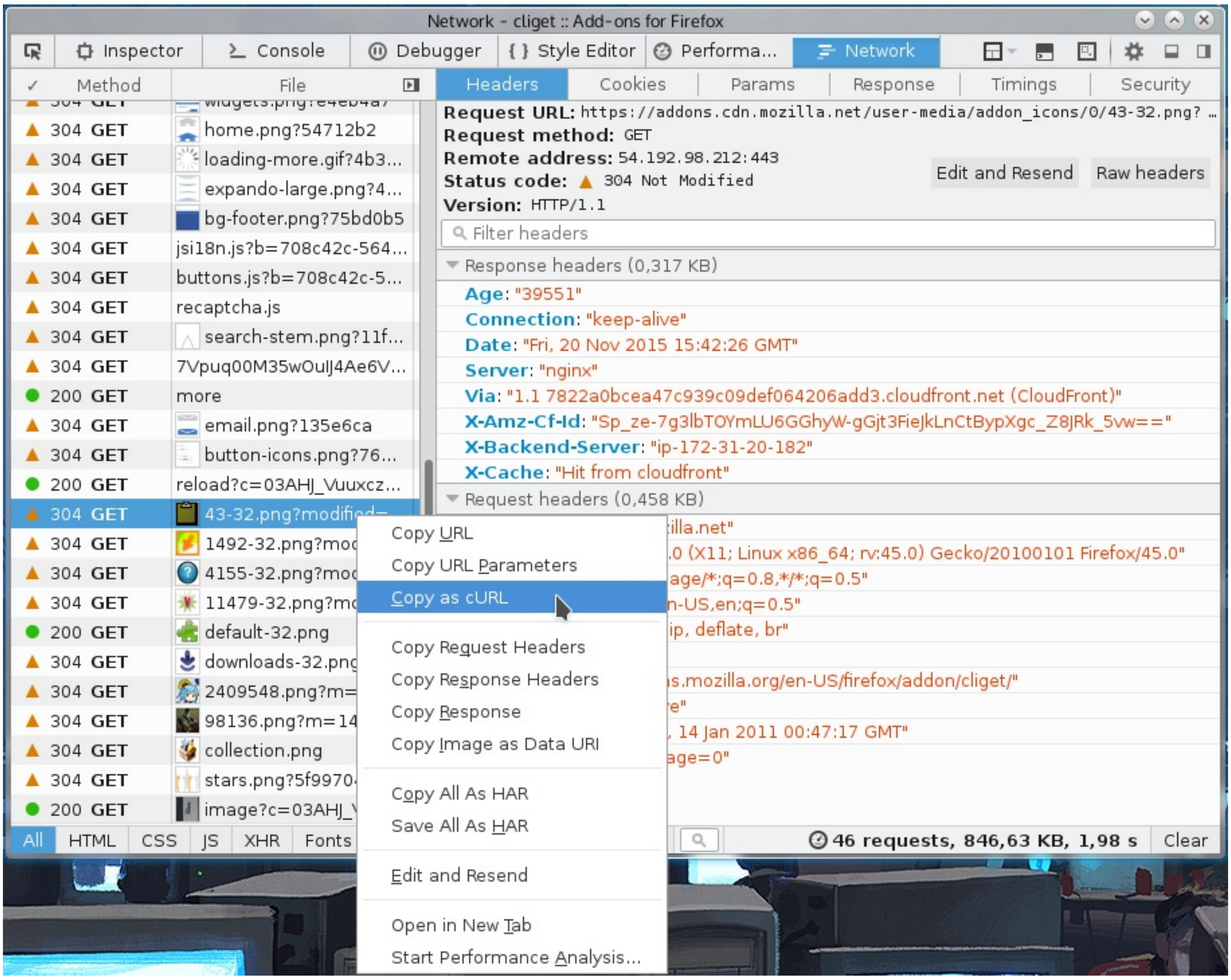
Copy as curl

Using curl to perform an operation a user just managed to do with his or her browser is one of the more common requests and areas people ask for help about.

How do you get a curl command line to get a resource, just like the browser would get it, nice and easy? Chrome, Firefox and Safari all have this feature.

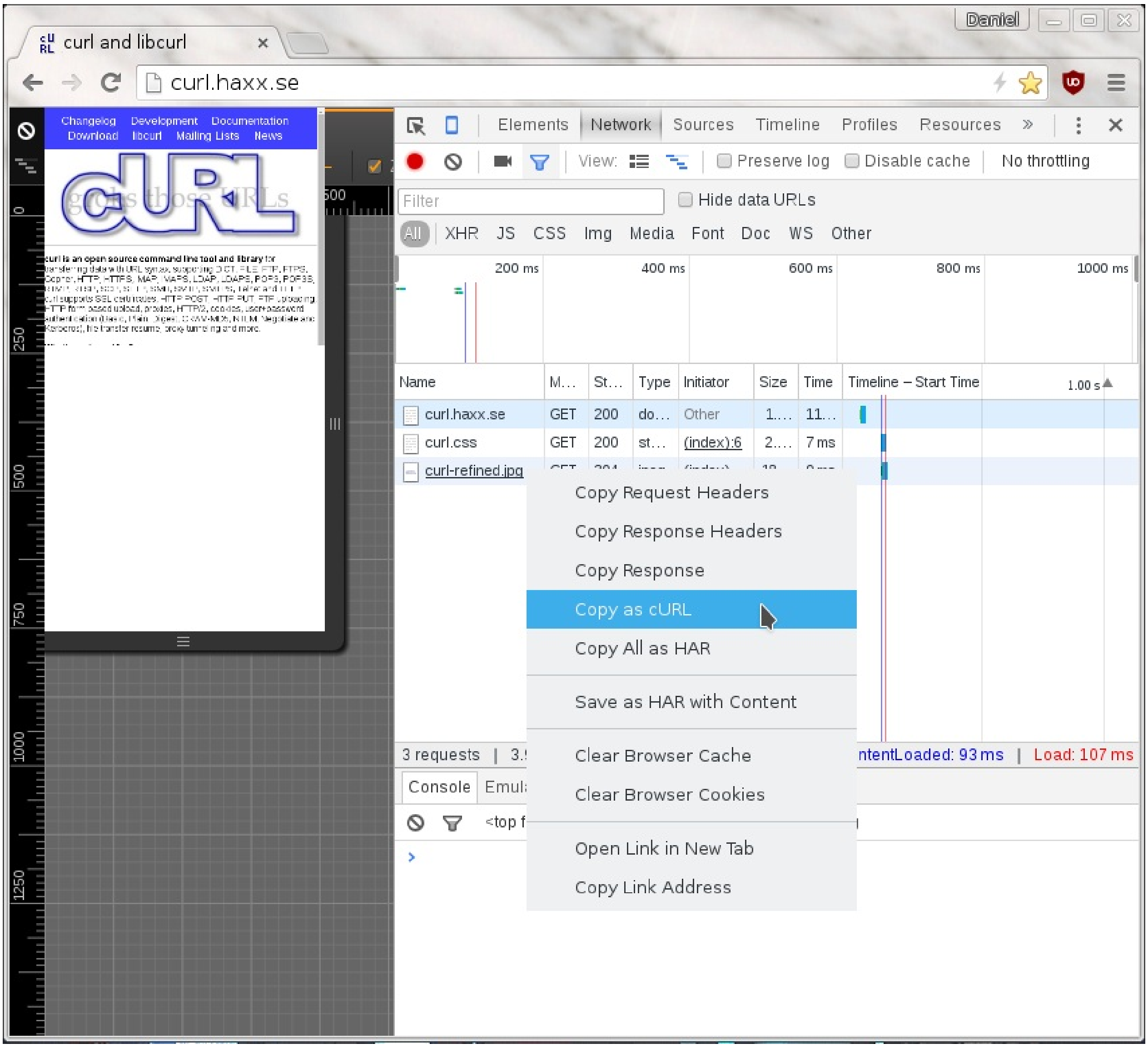
From Firefox

You get the site shown with Firefox's network tools. You then right-click on the specific request you want to repeat in the "Web Developer->Network" tool when you see the HTTP traffic, and in the menu that appears you select "Copy as cURL". Like this screenshot below shows. The operation then generates a curl command line to your clipboard and you can then paste that into your favorite shell window. This feature is available by default in all Firefox installations.



From Chrome

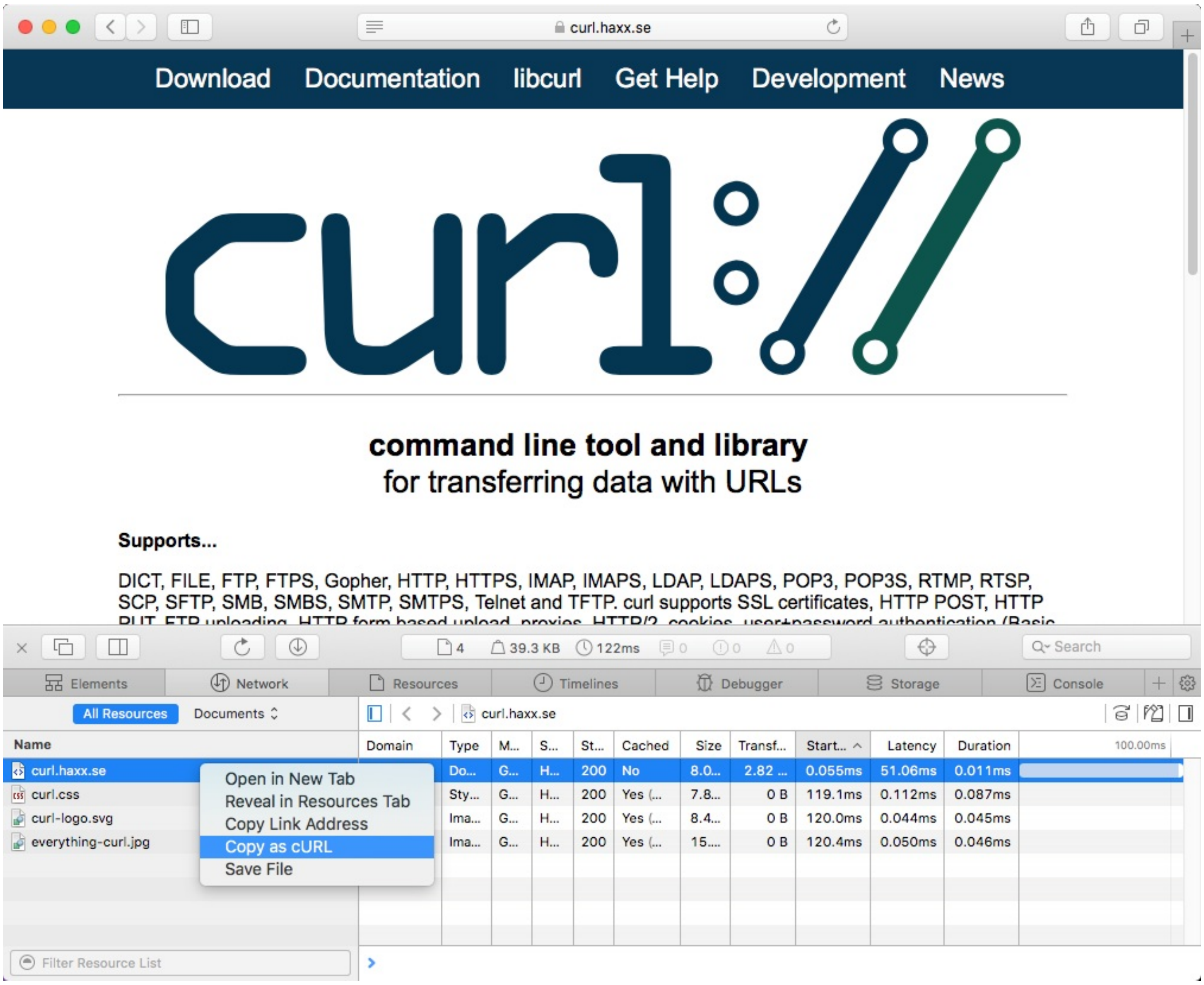
When you pop up the More tools->Developer mode in Chrome, and you select the Network tab you see the HTTP traffic used to get the resources of the site. On the line of the specific resource you are interested in, you right-click with the mouse and you select "Copy as cURL" and it will generate a command line for you in your clipboard. Paste that in a shell to get a curl command line that makes the transfer. This feature is available by default in all Chrome and Chromium installations.



From Safari

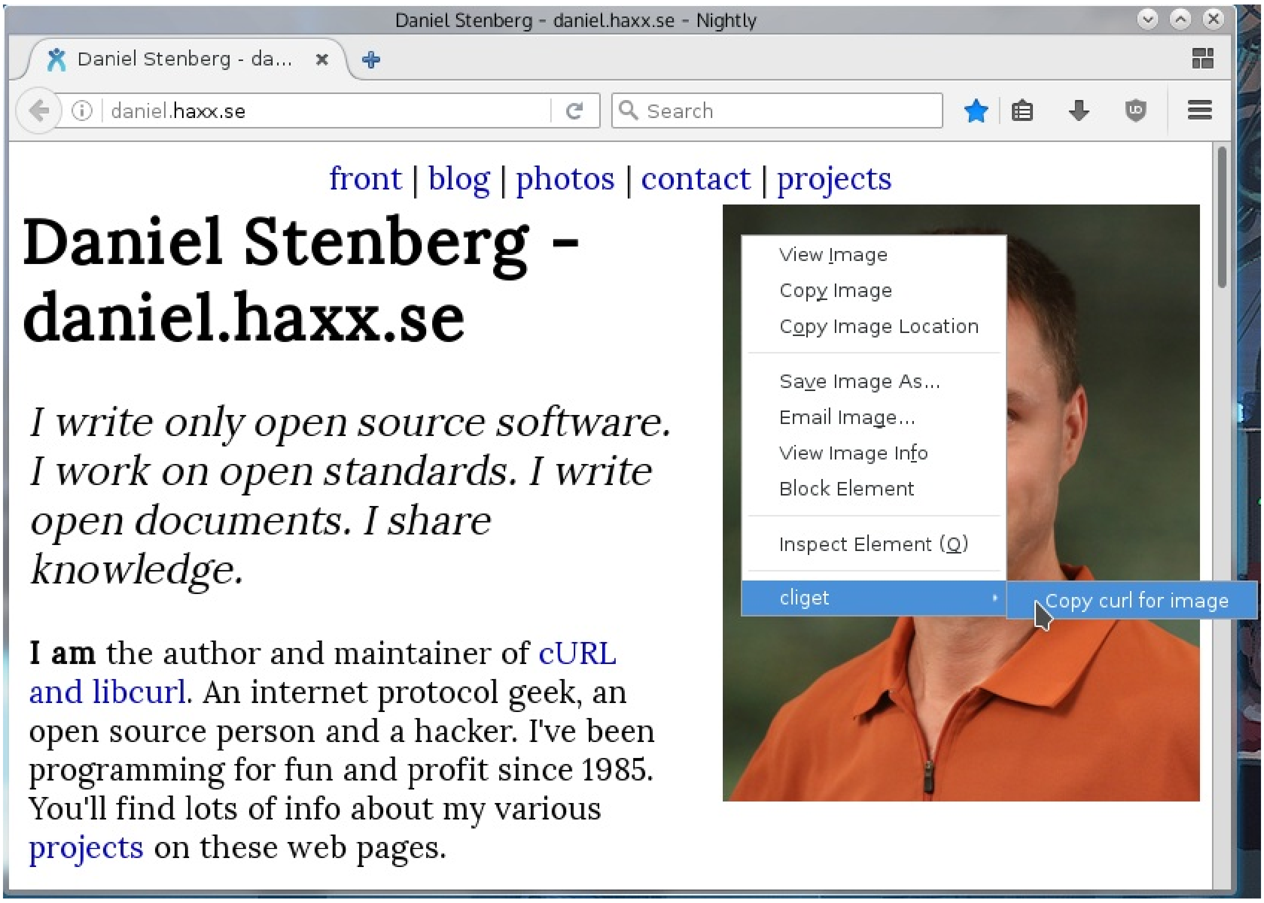
In Safari, the "development" menu isn't visible until you go into preferences->Advanced and enable it. But once you've done that, you can select "Show web inspector" in that development menu and get to see a new console pop up that is similar to the development tools of Firefox and Chrome.

Select the network tab, reload the web page and then you can right click the particular resources that you want to fetch with curl, as if you did it with Safari..



On Firefox, without using the devtools

If this is something you would like to get done more often, you probably find using the developer tools a bit inconvenient and cumbersome to pop up just to get the command line copied. Then cliget is the perfect add-on for you as it gives you a new option in the right-click menu, so you can get a quick command line generated really quickly, like this example when I right-click an image in Firefox:



Not perfect

These methods all give you a command line to reproduce their HTTP transfers, but you will also learn they they are still often not the perfect solution to your problems. Why? Well mostly because these tools are written to rerun the *exact* same request that you copied, while you often want to rerun the same logic but not sending an exact copy of the same cookies and file contents etc.

These tools will give you command lines with static and fixed cookie contents to send in the request, because that is the contents of the cookies that were sent in the browser's requests. You will most likely want to rewrite the command line to dynamically adapt to whatever the content is in the cookie that the server told you in a previous response. And so on.

The copy as curl functionality is also often notoriously bad at using `-F` and instead they provide handcrafted `--data-binary` solutions including the mime separator strings etc.

curl examples

TBD

Fetch many variations on a URL

TBD

Follow redirects automatically

TBD

Impersonating a specific web browser

TBD

Issuing a web search

TBD

Tell the server where you didn't come from

TBD

Maintain state with cookies

TBD

Login to a web service with POST

TBD

Upload a file as with a HTML form

TBD

How to HTTP with curl

In all user surveys and during all curl's lifetime, HTTP has been the most important and most frequently used protocol that curl supports. This chapter will explain how to do effective HTTP transfers and general fiddling with curl.

This will mostly work the same way for HTTPS, as they are really the same thing under the hood, as HTTPS is HTTP with an extra security TLS layer. See also the specific HTTPS section below.

HTTP methods

In every HTTP request, there's a method. Sometimes called a verb. The most commonly used ones are GET, POST, HEAD and PUT.

Normally however you don't specify the method in the command line, but instead the exact method used depends on the specific options you use. GET is default, using `-d` or `-F` makes it a POST, `-I` generates a HEAD and `-T` sends a PUT.

More about this in the Modify the HTTP request section.

Scripting browser-like tasks

TBD

HTTP protocol basics

(This assumes you have read the Network and protocols section or are otherwise already familiar with protocols.)

HTTP is a protocol that is easy to learn the basics of. A client connects to a server—and it is always the client that takes the initiative—sends a request and receives a response. Both the request and the response consist of headers and a body. There can be little or a lot of information going in both directions.

An HTTP request sent by a client starts with a request line, followed by headers and then optionally a body. The most common HTTP request is probably the GET request which asks the server to return a specific resource, and this request does not contain a body.

When a client connects to 'example.com' and asks for the '/' resource, it sends a GET without a request body:

```
GET / HTTP/1.1
User-agent: curl/2000
Host: example.com
```

…the server could respond with something like below, with response headers and a response body ('hello'). The first line in the response also contains the response code and the specific version the server supports:

```
HTTP/1.1 200 OK
Server: example-server/1.1
Content-Length: 5
Content-Type: plain/text

hello
```

If the client would instead send a request with a small request body ('hello'), it could look like this:

```
POST / HTTP/1.1
Host: example.com
User-agent: curl/2000
Content-Length: 5

hello
```

A server always responds to an HTTP request unless something is wrong.

The URL converted to a request

So when a HTTP client is given a URL to operate on, that URL is then used, picked apart and those parts are used in various places in the outgoing request to the server. Let's take the an example URL:

```
https://www.example.com/path/to/file
```

- **https** means that curl will use TLS to the remote port 443 (which is the default port number when no specified is used in the URL).
- **www.example.com** is the host name that curl will resolve to one or more IP address to connect to. This host name will also be used in the HTTP request in the `Host:` header.
- **/path/to/file** is used in the HTTP request to tell the server which exact document/resources curl wants to fetch

--path-as-is

The path part of the URL is the part that starts with the first slash after the host name and ends either at the end of the URL or at a '?' or '#' (roughly speaking).

If you include substrings including `/../` or `/./` in the path, curl will automatically squash them before the path is sent to the server, as is dictated by standards and how such strings tend to work in local file systems. The `/../` sequence will remove the previous section so that `/hello/sir/../` ends up just `/hello/` and `/./` is simply removed so that `/hello/./sir/` becomes `/hello/sir/`.

To *prevent* curl from squashing those magic sequences before they are sent to the server and thus allow them through, the `--path-as-is` option exists.

HTTP responses

When an HTTP client talks HTTP to a server, the server *will* respond with an HTTP response message or curl will consider it an error and returns 52 with the error message "Empty reply from server".

Size of an HTTP response

An HTTP response has a certain size and curl needs to figure it out. There are several different ways to signal the end of an HTTP response but the most basic way is to use the `Content-Length:` header in the response and with that specify the exact number of bytes in the response body.

Some early HTTP server implementations had problems with file sizes greater than 2GB and wrongly managed to send Content-Length: headers with negative sizes or otherwise just plain wrong data. curl can be told to ignore the Content-Length: header completely with `--ignore-content-length`. Doing so may have some other negative side-effects but should at least let you get the data.

HTTP response codes

An HTTP transfer gets a 3 digit response code back in the first response line. The response code is the server's way of giving the client a hint about how the request was handled.

It is important to note that curl does not consider it an error even if the response code would indicate that the requested document couldn't be delivered (or similar). curl considers a successful sending and receiving of HTTP to be good.

The first digit of the HTTP response code is a kind of "error class":

- 1xx: transient response, more is coming
- 2xx: success
- 3xx: a redirect
- 4xx: the client asked for something the server couldn't/wouldn't deliver
- 5xx: there's problem in the server

Remember that you can use curl's `--write-out` option to extract the response code. See the --write-out section.

CONNECT response codes

Since there can be a HTTP request and a separate CONNECT request in the same curl transfer, we often separate the CONNECT response (from the proxy) from the remote server's HTTP response.

The CONNECT is also an HTTP request so it gets response codes in the same numeric range and you can use `--write-out` to extract that code as well.

Chunked transfer encoding

An HTTP 1.1 server can decide to respond with a "chunked" encoded response, a feature that wasn't present in HTTP 1.0.

When sending a chunked response, there's no Content-Length: for the response to indicate its size. Instead, there's a `Transfer-Encoding: chunked` header that tells curl there's chunked data coming and then in the response body, the data comes in a series of "chunks". Every individual chunk starts with the size of that particular chunk (in hexadecimal), then a newline and then the contents of the chunk. This is repeated over and over until the end of the response, which is signalled with a zero sized chunk. The point of this sort of response is for the client to be able to figure out when the responses has ended even though the server didn't know the full size before it started to send it. This is usually the case when the response is dynamic and generated at the point when the request comes.

Clients like curl will, of course, decode the chunks and not show the chunk sizes to users.

Gzipped transfers

Responses over HTTP can be sent in compressed format. This is most commonly done by the server when it includes a `Content-Encoding: gzip` in the response as a hint to the client. Compressed responses make a lot of sense when either static resources are sent (that were compressed at a previous moment in time) or even in run-time when there's more CPU power available than bandwidth. Sending a much smaller amount of data is often preferred.

You can ask curl to both ask for compressed content *and* automatically and transparently uncompress gzipped data when receiving content encoded gzip (or in fact any other compression algorithm that curl understands) by using `--compressed`:

```
curl --compressed http://example.com/
```

Transfer encoding

A less common feature used with transfer encoding is compression.

Compression in itself is common. Over time the dominant and web compatible way to do compression for HTTP has become to use `Content-Encoding` as described in the section above. But HTTP was originally intended and specified to allow transparent compression as a transfer encoding, and curl supports this feature.

The client then simply asks the server to do compression transfer encoding and if acceptable, it will respond with a header indicating that it will and curl will then transparently uncompress that data on arrival. A user enables asking for compressed transfer encoding with `--tr-encoding`:

```
curl --tr-encoding http://example.com/
```

It should be noted that not many HTTP servers in the wild support this.

Pass on transfer encoding

In some situations you may want to use curl as some sort of proxy or other in between software. In those cases, curl's way to deal with transfer-encoding headers and then decoding the actual data transparently may not be desired, if the end receiver *also* expects to do the same.

You can then ask curl to pass on the received data, without decoding it. That means passing on the sizes in the chunked encoding format or the compressed format when compressed transfer encoding is used etc.

```
curl --raw http://example.com/
```

HTTP authentication

Each HTTP request can be made authenticated. If a server or a proxy wants the user to provide proof that they have the correct credentials to access a URL or perform an action, it can send back a HTTP response code that informs the client that it needs to provide a correct HTTP authentication header in the request to be allowed.

A server that requires authentication sends back a 401 response code and an associated `WWW-Authenticate:` header that lists all the authentication methods that the server supports.

An HTTP proxy that requires authentication sends back a 407 response code and an associated `Proxy-Authenticate:` header that lists all the authentication methods that the proxy supports.

It might be worth to note that most web sites of today do not require HTTP authentication for login etc, but they will instead ask users to login on web pages and then the browser will issue a POST with the user and password etc, and then subsequently maintain cookies for the session.

To tell curl to do an authenticated HTTP request, you use the `-u, --user` option to provide user name and password (separated with a colon). Like this:

```
curl --user daniel:secret http://example.com/
```

This will make curl use the default "Basic" HTTP authentication method. Yes, it is actually called Basic and it is truly very basic. To explicitly ask for the basic method, use `--basic`.

The Basic authentication method sends the user name and password in clear text over the network (base64 encoded) and should be avoided for HTTP transport.

When asking to do a HTTP transfer using a single (specified or implied), authentication method, curl will insert the authentication header already in the first request on the wire.

If you'd rather have curl first *test* if the authentication is really required, you can ask curl to figure that out and then automatically use the most safe method it knows about with `--anyauth`. This makes curl try the request unauthenticated, and then switch over to authentication if necessary:

```
curl --anyauth --user daniel:secret http://example.com/
```

and the same concept works for HTTP operations that may require authentication:

```
curl --proxy-anyauth --proxy-user daniel:secret http://example.com/ \
     --proxy http://proxy.example.com:80/
```

curl typically (a little depending on how it was built) speaks several other authentication methods as well, including Digest, Negotiate and NTLM. You can ask for those methods too specifically:

```
curl --digest --user daniel:secret http://example.com/
curl --negotiate --user daniel:secret http://example.com/
curl --ntlm --user daniel:secret http://example.com/
```

HTTP ranges

What if the client only wants the first 200 bytes out of a remote resource or perhaps 300 bytes somewhere in the middle? The HTTP protocol allows a client to ask for only a specific data range. The client asks the server for the specific range with a start offset and an end offset. It can even combine things and ask for several ranges in the same request by just listing a bunch of pieces next to each other. When a server sends back multiple independent pieces to answer such a request, you will get them separated with mime boundary strings and it will be up to the user application to handle that accordingly. curl will not further separate such a response.

However, a byte range is only a request to the server. It does not have to respect the request and in many cases, like when the server automatically generates the contents on the fly when it is being asked, it will simply refuse to do it and it then instead responds with the full contents anyway.

You can make curl ask for a range with `-r` or `--range`. If you want the first 200 bytes out of something:

```
curl -r 0-199 http://example.com
```

Or everything in the file starting from index 200:

```
curl -r 200- http://example.com
```

Get 200 bytes from index 0 *and* 200 bytes from index 1000:

```
curl -r 0-199,1000-199 http://example.com/
```

HTTP versions

As any other Internet protocol, the HTTP protocol has kept evolving over the years and now there are clients and servers distributed over the world and over time that speak different versions with varying levels of success. So in order to get libcurl to work with the URLs you pass in libcurl offers ways for you to specify which HTTP version that request and transfer should use. libcurl is designed in a way so that it tries to use the most common, the most sensible if you want, default values first but sometimes that isn't enough and then you may need to instruct libcurl what to do.

Since perhaps mid 2016, curl will default to use HTTP/1.1 for HTTP servers. If you connect to HTTPS and you have a libcurl that has HTTP/2 abilities built-in, curl will attempt to use HTTP/2 automatically or fall down to 1.1 in case the negotiation failed. Non-HTTP/2 capable curls get 1.1 over HTTPS by default.

If the default isn't good enough for your transfer, the `CURLOPT_HTTP_VERSION` option is there for you.

Option	Description
[default]	fill in
--http1.0	fill in
--http1.1	fill in
--http2	fill in
--http2-prior-knowledge	fill in

HTTPS

HTTPS is in effect Secure HTTP. The "secure" part means that the TCP transport layer is enhanced to provide authentication, privacy (encryption) and data integrity by the use of TLS.

See the Using TLS section for in-depth details on how to modify and tweak the TLS details in a HTTPS transfer.

HTTP POST

POST is the HTTP method that was invented to send data to a receiving web application, and it is how most common HTML forms on the web works. It usually sends a chunk of relatively small amounts of data to the receiver.

When the data is sent by a browser after data have been filled in a form, it will send it "URL encoded", as a serialized name=value pairs separated with ampersand symbols ('&'). You send such data with curl's `-d` or `--data` option like this:

```
curl -d 'name=admin&shoesize=12' http://example.com/
```

When specifying multiple `-d` options on the command line, curl will concatenate them and insert ampersands in between, so the above example could also be made like this:

```
curl -d name=admin -d shoesize=12 http://example.com/
```

If the amount of data to send isn't really fit to put in a mere string on the command line, you can also read it off a file name in standard curl style:

```
curl -d @filename http://example.com
```

Content-Type

POSTing with curl's -d option will make it include a default header that looks like `Content-Type: application/x-www-form-urlencoded`. That's what your typical browser will use for a plain POST.

Many receivers of POST data don't care about or check the Content-Type header.

If that header is not good enough for you, you should, of course, replace that and instead provide the correct one. Such as if you POST JSON to a server and want to more accurately tell the server about what the content is:

```
curl -d '{json}' -H 'Content-Type: application/json' https://example.com
```

POSTing binary

When reading from a file, `-d` will strip out carriage return and newlines. Use `--data-binary` if you want curl to read and use the given file in binary exactly as given:

```
curl --data-binary @filename http://example.com/
```

URL encoding

Percent-encoding, also known as URL encoding, is technically a mechanism for encoding data so that it can appear in URLs. This encoding is typically used when sending POSTs with the `application/x-www-form-urlencoded` content type, such as the ones curl sends with `--data` and `--data-binary` etc.

The command-line options mentioned above all require that you provide properly encoded data, data you need to make sure already exists in the right format. While that gives you a lot of freedom, it is also a bit inconvenient at times.

To help you send data you haven't already encoded, curl offers the `--data-urlencode` option. This option offers several different ways to URL encode the data you give it.

You use it like `--data-urlencode data` in the same style as the other --data options. To be CGI-compliant, the **data** part should begin with a name followed by a separator and a content specification. The **data** part can be passed to curl using one of the following syntaxes:

- "content": This will make curl URL encode the content and pass that on. Just be careful so that the content doesn't contain any = or @ symbols, as that will then make the syntax match one of the other cases below!
- "=content": This will make curl URL encode the content and pass that on. The initial '=' symbol is not included in the data.
- "name=content": This will make curl URL encode the content part and pass that on. Note that the name part is expected to be URL encoded already.
- "@filename": This will make curl load data from the given file (including any newlines), URL encode that data and pass it on in the POST.
- "name@filename": This will make curl load data from the given file (including any newlines), URL encode that data and pass it on in the POST. The name part gets an equal sign appended, resulting in name=urlencoded-file-content. Note that the name is expected to be URL encoded already.

As an example, you could POST a name to have it encoded by curl:

```
curl --data-urlencode "name=John Doe (Junior)" http://example.com
```

…which would send the following data in the actual request body:

```
name=John%20Doe%20%28Junior%29
```

If you store the string `John Doe (Junior)` in a file named `contents.txt`, you can tell curl to send that contents URL encoded using the field name 'user' like this:

```
curl --data-urlencode user@contents.txt http://example.com
```

In both these examples above, the field name is not URL encoded but is passed on as-is. If you want to URL encode the field name as well, like if you want to pass on a field name called "user name", you can ask curl to encode the entire string by prefixing it with an equals sign (that will not get sent):

```
curl --data-urlencode "=user name=John Doe (Junior)" http://example.com
```

Convert that to a GET

A little convenience feature that could be suitable to mention in this context (even though it isn't for POSTing), is the `-G` or `--get` option, which takes all data you have specified with the different `-d` variants and appends that data on the right end of the URL separated with a '?' and then makes curl send a GET instead.

This option makes it easy to switch between POSTing and GETing a form, for example.

Expect 100-continue

HTTP has no proper way to stop an ongoing transfer (in any direction) and still maintain the connection. So, if we figure out that the transfer had better stop after the transfer has started, there are only two ways to proceed: cut the connection and pay the price of reestablishing the connection again for the next request, or keep the transfer going and waste bandwidth but be able to reuse the connection next time.

One example of when this can happen is when you send a large file over HTTP, only to discover that the server requires authentication and immediately sends back a 401 response code.

The mitigation that exists to make this scenario less frequent is to have curl pass on an extra header, `Expect: 100-continue`, which gives the server a chance to deny the request before a lot of data is sent off. curl sends this Expect: header by default if the POST it will do is known or suspected to be larger than just minuscule. curl also does this for PUT requests.

When a server gets a request with an 100-continue and deems the request fine, it will respond with a 100 response that makes the client continue. If the server doesn't like the request, it sends back response code for the error it thinks it is.

Unfortunately, lots of servers in the world don't properly support the Expect: header or don't handle it correctly, so curl will only wait 1000 milliseconds for that first response before it will continue anyway.

Those are 1000 wasted milliseconds. You can then remove the use of Expect: from the request and avoid the waiting with `-H`:

```
curl -H Expect: -d "payload to send" http://example.com
```

In some situations, curl will inhibit the use of the Expect header if the content it is about to send is very small (like below one kilobyte), as having to "waste" such a small chunk of data is not considered much of a problem.

Chunked encoded POSTs

When talking to a HTTP 1.1 server, you can tell curl to send the request body without a `Content-Length:` header upfront that specifies exactly how big the POST is. By insisting on curl using chunked Transfer-Encoding, curl will send the POST "chunked" piece by piece in a special style that also sends the size for each such chunk as it goes along.

You send a chunked POST with curl like this:

```
curl -H "Transfer-Encoding: chunked" -d "payload to send" http://example.com
```

Hidden form fields

This chapter has explained how sending a post with `-d` is the equivalent of what a browser does when an HTML form is filled in and submitted.

Submitting such forms is a very common operation with curl; effectively, to have curl fill in a web form in an automated fashion.

If you want to submit a form with curl and make it look as if it has been done with a browser, it is important to provide all the input fields from the form. A very common method for web pages is to set a few hidden input fields to the form and have them assigned values directly in the HTML. A successful form submission, of course, also includes those fields and in order to do that automatically you may be forced to first download the HTML page that holds the form, parse it, and extract the hidden field values so that you can send them off with curl.

Figure out what a browser sends

A very common shortcut is to simply fill in the form with your browser and submit it and check in the browser's network development tools exactly what it sent.

A slightly different way is to save the HTML page containing the form, and then edit that HTML page to redirect the 'action=' part of the form to your own server or a test server that just outputs exactly what it gets. Completing that form submission will then show you exactly how a browser sends it.

A third option is, of course, to use a network capture tool such as Wireshark to check exactly what is sent over the wire. If you are working with HTTPS, you can't see form submissions in clear text on the wire but instead you need to make sure you can have Wireshark extract your TLS private key from your browser. See the Wireshark documentation for details on doing that.

JavaScript and forms

A very common mitigation against automated "agents" or scripts using curl is to have the page with the HTML form use JavaScript to set values of some input fields, usually one of the hidden ones. Often, there's some JavaScript code that executes on page load or when the submit button is pressed which sets a magic value that the server then can verify before it considers the submission to be valid.

You can usually work around that by just reading the JavaScript code and redoing that logic in your script. Using the above mentioned tricks to check exactly what a browser sends is then also a good help.

HTTP multipart formposts

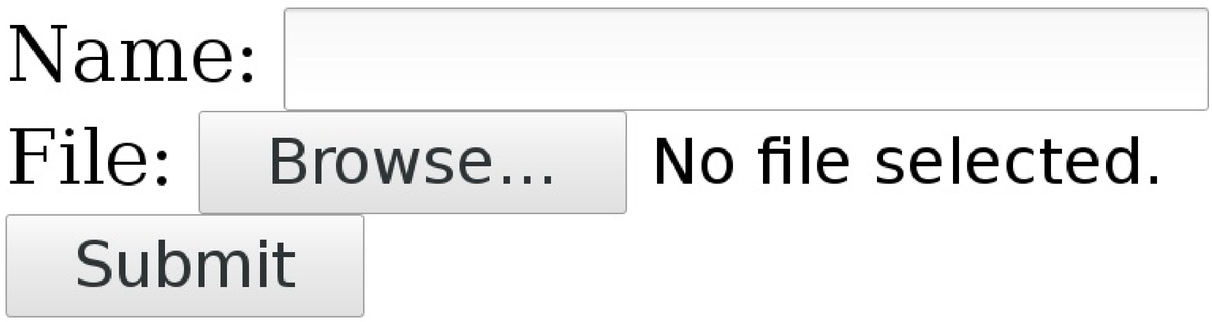
A multipart formpost is what an HTTP client sends when an HTML form is submitted with *enctype* set to "multipart/form-data".

It is an HTTP POST request sent with the request body specially formatted as a series of "parts", separated with MIME boundaries.

An example piece of HTML would look like this:

```
<form action="submit.cgi" method="post" enctype="multipart/form-data">
   Name: <input type="text" name="person"><br>
   File: <input type="file" name="secret"><br>
   <input type="submit" value="Submit">
</form>
```

Which could look something like this in a web browser:



A user can fill in text in the 'Name' field and by pressing the 'Browse' button a local file can be selected that will be uploaded when 'Submit' is pressed.

Sending such a form with curl

With curl, you add each separate multipart with one `-F` (or `--form`) flag and you then continue and add one -F for every input field in the form that you want to send.

The above small example form has two parts, one named 'person' that is a plain text field and one named 'secret' that is a file.

Send your data to that form like this:

```
curl -F person=anonymous -F secret=@file.txt http://example.com/submit.cgi
```

The HTTP this generates

The **action** specifies where the POST is sent. **method** says it is a POST and **enctype** tells us it is a multipart formpost.

With the fields filled in as shown above, curl generates and sends these HTTP request headers to the host example.com:

```
POST /submit.cgi HTTP/1.1
Host: example.com
User-Agent: curl/7.46.0
Accept: */*
Content-Length: 313
Expect: 100-continue
Content-Type: multipart/form-data; boundary=------------------------d74496d66958873e
```

Content-Length, of course, tells the server how much data to expect. This example's 313 bytes is really small.

The **Expect** header is explained in the HTTP POST chapter.

The **Content-Type** header is a bit special. It tells that this is a multipart formpost and then it sets the "boundary" string. The boundary string is a line of characters with a bunch of random digits somewhere in it, that serves as a separator between the different parts of the form that will be submitted. The particular boundary you see in this example has the random part `d74496d66958873e` but you will, of course, get something different when you run curl (or when you submit such a form with a browser).

So after that initial set of headers follows the request body

```
--------------------------d74496d66958873e
Content-Disposition: form-data; name="person"

anonymous
--------------------------d74496d66958873e
Content-Disposition: form-data; name="secret"; filename="file.txt"
Content-Type: text/plain

contents of the file
--------------------------d74496d66958873e--
```

Here you clearly see the two parts sent, separated with the boundary strings. Each part starts with one or more headers describing the individual part with its name and possibly some more details. Then after the part's headers come the actual data of the part, without any sort of encoding.

The last boundary string has two extra dashes `--` appended to signal the end.

Content-Type

POSTing with curl's -F option will make it include a default Content-Type header in its request, as shown in the above example. This says `multipart/form-data` and then specifies the MIME boundary string. That content-type is the default for multipart formposts but you can, of course, still modify that for your own commands and if you do, curl is clever enough to still append the boundary magic to the replaced header. You can't really alter the boundary string, since curl needs that for producing the POST stream.

To replace the header, use `-H` like this:

```
curl -F 'name=Dan' -H 'Content-Type: multipart/magic' https://example.com
```

Converting an HTML form

TBD

-d vs -F

Previous chapters talked about regular POST and multipart formpost, and in your typical command lines you do them with `-d` or `-F`.

But when do you use which of them?

As described in the chapters mentioned above, both these options send the specified data to the server. The difference is in how the data is formatted over the wire. Most of the time, the receiving end is written to expect a specific format and it expects that the sender formats and sends the data correctly. A client cannot just pick a format of its own choice.

HTML web forms

When we are talking browsers and HTML, the standard way is to offer a form to the user that sends off data when the form has been filled in. The `<form>` tag is what makes one of those appear on the web page. The tag instructs the browser how to format its POST. If the form tag includes `enctype=multipart/form-data`, it tells the browser to send the data as a multipart formpost which you make with curl's `-F` option. This method is typically used when the form includes a `<input type=file>` tag, for file uploads.

The default `enctype` used by forms, which is rarely spelled out in HTML since it is default, is `application/x-www-form-urlencoded`. It makes the browser "URL encode" the input as name=value pairs with the data encoded to avoid unsafe character. We often refer to that as a regular POST, and you perform one with curl's `-d` and friends.

POST outside of HTML

POST is a regular HTTP method and there is no requirement that it be triggered by HTML or involve a browser. Lots of services, APIs and other systems allow you to pass in data these days in order to get things done.

If these services expect plain "raw" data or perhaps data formatted as JSON or similar, you want the regular POST approach. curl's `-d` option won't alter or encode the data at all but will just send exactly what you tell it to. Just pay attention to -d's default Content-Type as that might not be what you want.

HTTP redirects

The “redirect” is a fundamental part of the HTTP protocol. The concept was present and is documented already in the first spec (RFC 1945), published in 1996, and it has remained well-used ever since.

A redirect is exactly what it sounds like. It is the server sending back an instruction to the client instead of giving back the contents the client wanted. The server basically says “go look over *here* instead for that thing you asked for“.

But not all redirects are alike. How permanent is the redirect? What request method should the client use in the next request?

All redirects also need to send back a `Location:` header with the new URI to ask for, which can be absolute or relative.

Permanent and temporary

Is the redirect meant to last or just remain valid for now? If you want a GET to permanently redirect users to resource B with another GET, send back a 301. It also means that the user-agent (browser) is meant to cache this and keep going to the new URI from now on when the original URI is requested.

The temporary alternative is 302. Right now the server wants the client to send a GET request to B, but it shouldn't cache this but keep trying the original URI when directed to it next time.

Note that both 301 and 302 will make browsers do a GET in the next request, which possibly means changing the method if it started with a POST (and only if POST). This changing of the HTTP method to GET for 301 and 302 responses is said to be “for historical reasons”, but that’s still what browsers do so most of the public web will behave this way.

In practice, the 303 code is very similar to 302. It will not be cached and it will make the client issue a GET in the next request. The differences between a 302 and 303 are subtle, but 303 seems to be more designed for an “indirect response” to the original request rather than just a redirect.

These three codes were the only redirect codes in the HTTP/1.0 spec.

curl however, doesn't remember or cache any redirects at all so to it, there's really no difference between permanent and temporary redirects.

Tell curl to follow redirects

In curl's tradition of only doing the basics unless you tell it differently, it doesn't follow HTTP redirects by default. Use the `-L, --location` to tell it to do that.

When following redirects is enabled, curl will follow up to 50 redirects by default. There's a maximum limit mostly to avoid the risk of getting caught in endless loops. If 50 isn't sufficient for you, you can change the maximum number of redirects to follow with the `--max-redirs` option.

GET or POST?

All three of these response codes, 301 and 302/303, will assume that the client sends a GET to get the new URI, even if the client might have sent a POST in the first request. This is very important, at least if you do something that doesn't use GET.

If the server instead wants to redirect the client to a new URI and wants it to send the same method in the second request as it did in the first, like if it first sent POST it'd like it to send POST again in the next request, the server would use different response codes.

To tell the client "the URI you sent a POST to, is permanently redirected to B where you should instead send your POST now and in the future", the server responds with a 308. And to complicate matters, the 308 code is only recently defined (the spec was published in June 2014) so older clients may not treat it correctly! If so, then the only response code left for you is…

The (older) response code to tell a client to send a POST also in the next request but temporarily is 307. This redirect will not be cached by the client though, so it'll again post to A if requested to again. The 307 code was introduced in HTTP/1.1.

Oh, and redirects work the exact same way in HTTP/2 as they do in HTTP/1.1.

	Permanent	Temporary
Switch to GET	301	302 and 303
Keep original method	308	307

Decide what method to use in redirects

It turns out that there are web services out there in the world that want a POST sent to the original URL, but are responding with HTTP redirects that use a 301, 302 or 303 response codes and *still* want the HTTP client to send the next request as a POST. As explained

above, browsers won't do that and neither will curl—by default.

Since these setups exist, and they're actually not terribly rare, curl offers options to alter its behavior.

You can tell curl to not change the non-GET request method to GET after a 30x response by using the dedicated options for that: `--post301` , `--post302` and `--post303` . If you are instead writing a libcurl based application, you control that behavior with the `CURLOPT_POSTREDIR` option.

Redirecting to other host names

When you use curl you may provide credentials like user name and password for a particular site, but since a HTTP redirect might very well move away to a different host curl limits what it sends away to other hosts than the original within the same "transfer".

So if you want the credentials to also get sent to the following host names even though they are not the same as the original—presumably because you trust them and know that there's no harm in doing that—you can tell curl that it is fine to do so by using the `--location-trusted` option.

Non-HTTP redirects

Browsers support more ways to do redirects that sometimes make life complicated to a curl user as these methods are not supported or recognized by curl.

HTML redirects

If the above wasn't enough, the web world also provides a method to redirect browsers by plain HTML. See the example `<meta>` tag below. This is somewhat complicated with curl since curl never parses HTML and thus has no knowledge of these kinds of redirects.

```
<meta http-equiv="refresh" content="0; url=http://example.com/">
```

JavaScript redirects

The modern web is full of JavaScript and as you know, JavaScript is a language and a full run time that allows code to execute in the browser when visiting web sites.

JavaScript also provides means for it to instruct the browser to move on to another site—a redirect, if you will.

Modify the HTTP request

As described earlier, each HTTP transfer starts with curl sending a HTTP request. That request consists of a request line and a number of request headers, and this chapter details how you can modify all of those.

Request method

The first line of the request includes the *method* - sometimes also referred to as "the verb". When doing a simple GET request as this command line would do:

```
curl http://example.com/file
```

…the initial request line looks like this:

```
GET /file HTTP/1.1
```

You can tell curl to change the method into something else by using the `-X` or `--request` command-line options followed by the actual method name. You can, for example, send a `DELETE` instead of `GET` like this:

```
curl http://example.com/file -X DELETE
```

This command-line option only changes the text in the outgoing request, it does not change any behavior. This is particularly important if you, for example, ask curl to send a HEAD with `-X`, as HEAD is specified to send all the headers a GET response would get but *never* send a response body, even if the headers otherwise imply that one would come. So, adding `-X HEAD` to a command line that would otherwise do a GET will cause curl to hang, waiting for a response body that won't come.

When asking curl to perform HTTP transfers, it will pick the correct method based on the option so you should only very rarely have to explicitly ask for it with `-X`. It should also be noted that when curl follows redirects like asked to with `-L`, the request method set with `-X` will be sent even on the subsequent redirects.

Request target

In the example above, you can see how the path section of the URL gets turned into `/file` in the request line. That is called the "request target". That's the resource this request will interact with. Normally this request target is extracted from the URL and then used in the request and as a user you don't need to think about it.

In some rare circumstances, user may want to go creative and change this request target in ways that the URL doesn't really allow. For example, the HTTP OPTIONS method has a specially define request target for magic that concerns *the server* and not a specific path, and it uses `*` for that. Yes, a single asterisk. There's no way to specify a URL for this, so if you want to pass a single asterisk in the request target to a server, like for OPTIONS, you have to do it like this:

```
curl -X OPTIONS --request-target "*" http://example.com/
```

Anchors or fragments

A URL may contain an anchor, also known as a fragment, which is written with a pound sign and string at the end of the URL. Like for example `http://example.com/foo.html#here-it-is`. That fragment part, everything from the pound/hash sign to the end of the URL, is only intend for local use and will not be sent over the network. curl will simply strip that data off and discard it.

Customize headers

In a HTTP request, after the initial request-line, there will typically follow a number of request headers. That's a set of `name: value` pairs that ends with a blank line that separates the headers from the following request body (that sometimes is empty).

curl will by default and on its own account pass a few headers in requests, like for example `Host:`, `Accept:`, `User-Agent:` and a few others that may depend on what the user asks curl to do.

All headers set by curl itself can be overridden, replaced if you will, by the user. You just then tell curl's `-H` or `--header` the new header to use and it will then replace the internal one if the header field matches one of those headers, or it will add the specified header to the list of headers to send in the request.

To change the `Host:` header, do this:

```
curl -H "Host: test.example" http://example.com/
```

To add a `Elevator: floor-9` header, do this:

```
curl -H "Elevator: floor-9" http://example.com/
```

If you just want to delete an internally generated header, just give it to curl without a value, just nothing on the right side of the colon.

To switch off the `User-Agent:` header, do this:

```
curl -H "User-Agent:" http://example.com/
```

Finally, if you then truly want to add a header with no contents on the right side of the colon (which is a rare thing), the magic marker for that is to instead end the header field name with a *semicolon*. Like this:

```
curl -H "Empty;" http://example.com
```

Referer

When a user clicks on a link on a web page and the browser takes the user away to the next URL, it will send the new URL a "referer" header in the new request telling it where it came from. That is the referer header. And yes, referer is misspelled but that's how it is supposed to be!

With curl you set the referer header with `-e` or `--referer`, like this:

```
curl --referer http://comes-from.example.com https://www.example.com/
```

User-agent

The User-Agent is a header that each client can set in the request to inform the server which user-agent it is. Sometimes servers will look at this header and determine how to act based on its contents.

The default header value is 'curl/[version]', as in `User-Agent: curl/7.54.1` for curl version 7.54.1.

You can set any value you like, using the option `-A` or `--user-agent` plus the string to use or, as it's just a header, `-H "User-Agent: foobar/2000"`.

As comparison, a recent test version of Firefox on a Linux machine sent this User-Agent header:

```
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:58.0) Gecko/20100101 Firefox/58.0
```

--time-cond

TBD

PUT

The difference between a PUT and a POST is subtle. They are virtually identical transmissions except for the different method strings. Where POST is meant to pass on data to a remote resource, PUT is supposed to be the new version of that resource.

In that aspect, PUT is similar to good old standard file upload found in other protocols. You upload a new version of the resource with PUT. The URL identifies the resource and you point out the local file to put there:

```
curl -T localfile http://example.com/new/resource/file
```

…so -T will imply a PUT and tell curl which file to send off. But the similarities between POST and PUT also allows you to send a PUT with a string by using the regular curl POST mechanism using `-d` but asking for it to use a PUT instead:

```
curl -d "data to PUT" -X PUT http://example.com/new/resource/file
```

Cookies

HTTP cookies are key/value pairs that a client stores on the behalf of a server. They are sent back in subsequent requests to allow clients to keep state between requests. Remember that the HTTP protocol itself has no state but instead the client has to resend all data in subsequent requests that it wants the server to be aware of.

Cookies are set by the server with the `Set-Cookie:` header and with each cookie the server sends a bunch of extra properties that need to match for the client to send the cookie back. Like domain name and path and perhaps most important for how long the cookie should live on.

The expiry of a cookie is either set to a fixed time in the future (or to live a number of seconds) or it gets no expiry at all. A cookie without an expire time is called a "session cookie" and is meant to live for the duration of the "session" but not longer. A session in this aspect is typically thought to be the life time of the browser used to view a site. When you close the browser, you end your session. Doing HTTP operations with a command-line client that supports cookies begs the question of when a session really ends…

Cookie engine

The general concept of curl only doing the bare minimum unless you tell it differently makes it not acknowledge cookies by default. You need to switch on "the cookie engine" to make curl keep track of cookies it receives and then subsequently send them out on requests that have matching cookies.

You enable the cookie engine by asking curl to read or write cookies. If you tell curl to read cookies from a non-existing file, you will only switch on the engine but start off with an empty internal cookie store:

```
curl -b non-existing http://example.com
```

But just switching on the cookie engine, getting a single resource and then quitting would be pointless as curl would have no chance to actually send any cookies it received. Assuming the site in this example would set cookies and then do a redirect we would do:

```
curl -L -b non-existing http://example.com
```

Reading cookies from file

Starting off with a blank cookie store may not be desirable. Why not start off with cookies you stored in a previous fetch or that you otherwise acquired? The file format curl uses for cookies is called the Netscape cookie format because it was once the file format used by browsers and then you could easily tell curl to use the browser's cookies!

As a convenience, curl also supports a cookie file being a set of HTTP headers that set cookies. It's an inferior format but may be the only thing you have.

Tell curl which file to read the initial cookies from:

```
curl -L -b cookies.txt http://example.com
```

Remember that this only *reads* from the file. If the server would update the cookies in its response, curl would update that cookie in its in-memory store but then eventually throw them all away when it exits and a subsequent invocation of the same input file would use the original cookie contents again.

Writing cookies to file

The place where cookies are stored is sometimes referred to as the "cookie jar". When you enable the cookie engine in curl and it has received cookies, you can instruct curl to write down all its known cookies to a file, the cookie jar, before it exists. It is important to remember that curl only updates the output cookie jar on exit and not during its lifetime, no matter how long the handling of the given input takes.

You point out the cookie jar output with `-c` :

```
curl -c cookie-jar.txt http://example.com
```

`-c` is the instruction to *write* cookies to a file, `-b` is the instruction to *read* cookies from a file. Oftentimes you want both.

When curl writes cookies to this file, it will save all known cookies including those that are session cookies (without a given lifetime). curl itself has no notion of a session and it doesn't know when a session ends so it will not flush session cookies unless you tell it to.

New cookie session

Instead of telling curl when a session ends, in order to flush session cookies and with this basically signal to the server that we are starting a new session, curl features an option that lets the user decide when a new session begins.

A new cookie session means that all the session cookies will be thrown away. It is the equivalent of closing a browser and starting it up again.

Tell curl a new cookie session starts by using `-j, --junk-session-cookies`:

```
curl -j -b cookies.txt http://example.com/
```

HTTP/2

curl supports HTTP/2 for both HTTP:// and HTTPS:// URLs assuming that curl was built with the proper prerequisites. It will even default to using HTTP/2 when given a HTTPS URL since doing so implies no penalty and when curl is used with sites that don't support HTTP/2 the request will instead negotiate HTTP/1.1.

With HTTP:// URLs however, the upgrade to HTTP/2 is done with an `Upgrade:` header that may cause an extra round-trip and perhaps even more troublesome, a sizable share of old servers will return a 400 response when seeing such a header.

It should also be noted that some (most?) servers that support HTTP/2 for HTTP:// (which in itself isn't all servers) will not acknowledge the `Upgrade:` header on POST, for example.

To ask a server to use HTTP/2, just:

```
curl --http2 http://example.com/
```

If your curl doesn't support HTTP/2, that command line will return an error saying so. Running `curl -V` will show if your version of curl supports it.

If you by some chance already know that your server speaks HTTP/2 (for example, within your own controlled environment where you know exactly what runs in your machines) you can shortcut the HTTP/2 "negotiation" with `--http2-prior-knowledge`.

Multiplexing

One of the primary features in the HTTP/2 protocol is the ability to multiplex several logical stream over the same physical connection. When using the curl command-line tool, you cannot take advantage of that cool feature since curl is doing all its network requests in a strictly serial manner, one after the next, with the second only ever starting once the previous one has ended.

Hopefully, a future curl version will be enhanced to allow the use of this feature.

curl HTTP cheat sheet

online here

Verbose	Hide progress	extra info	Write output	Timeout
-v	-s	-w "format"	-O	-m
--trace-ascii			-o	
POST	**multipart**	**PUT**	**HEAD**	**custom**
-d "string"	-F name=value	-T	-I	-X "METHOD"
-d @file	-F name=@file			
Basic auth	**read cookies**	**write cookies**	**send cookies**	**user-agent**
-u user:password	-b	-c	-b "c=1; d=2"	-A "string"
Use proxy	**Headers, add/remove**	**follow redirs**	**gzip**	**insecure**
-x	-H "name: value"	-L	--compressed	-k
	-H "name:"			

Building and installing

The source code for this project is written in a way that allows it to get compiled and built on just about any operating system and platform, with as few restraints and requirements as possible.

If you have a 32bit (or larger) CPU architecture, if you have a C89 compliant compiler and if you have roughly a POSIX supporting sockets API, then you can most likely build curl and libcurl for your target system.

For the most popular platforms, the curl project comes with build systems already done and prepared to allow you to easily build it yourself.

There are also friendly people and organizations who put together binary packages of curl and libcurl and make them available for download. The different options will be explored below.

The latest version?

Looking at the curl web site at https://curl.haxx.se you can see the latest curl and libcurl version released from the project. That's the latest source code package you can get.

When you opt for a prebuilt and prepackaged version for your operating system or distribution of choice, you may not always find the latest version but you might have to either be satisfied with the latest version someone has packaged for your environment, or you need to build it yourself from source.

The curl project also provides info about the latest version in a somewhat more machine-readable format on this URL: `https://curl.haxx.se/info` .

off git!

Of course, when building from source you can also always opt to build the very latest version that exist in the git repository. It could however be a bit more fragile and probably requires slightly more attention to detail.

Installing prebuilt binaries

There are many friendly people and organizations who put together binary packages of curl and libcurl and make them available for download.

Many operating systems offer a "package repository" that is populated with software packages for you to install. From the curl download page you can also follow links to plain binary packages for popular operating systems.

Installing from your package repository

curl and libcurl have been around for a very long time and most Linux distributions, BSD versions and other *nix versions have package repositories that allow you to install curl packages.

The most common ones are described below.

apt

`apt` is a tool to install prebuilt packages on Debian Linux and Ubuntu Linux distributions and derivatives.

To install the curl command-line tool, you usually just

```
apt install curl
```

…and that then makes sure the dependencies are installed and usually libcurl is then also installed as an individual package.

If you want to build applications against libcurl, you need a development package installed to get the include headers and some additional documentation, etc. You can then select a libcurl with the TLS backend you prefer:

```
apt install libcurl4-openssl-dev
```

or

```
apt install libcurl4-nss-dev
```

or

```
apt install libcurl4-gnutls-dev
```

yum

With Redhat Linux and CentOS Linux derivatives, you use `yum` to install packages. Install the command-line tool with:

```
yum install curl
```

You install the libcurl development package (with include files and some docs, etc.) with this:

```
yum install curl-devel
```

(The Redhat family of Linux systems usually ship curl built to use NSS as TLS backend.)

nix

Nix is a package manager default to the NixOS distribution, but it can also be used on any Linux distribution.

In order to install command-line tool:

```
nix-env -i curl
```

homebrew

Homebrew is an OSX package manager. It does not come by default, but one can install it easily.

To install the command-line tool:

```
brew install curl
```

cygwin

TBD

Binary packages for Windows

TBD

Build from source

The curl project creates source code that can be built to produce the two products curl and libcurl. The conversion from source code to binaries is often referred to as "building". You build curl and libcurl from source.

The curl project doesn't provide any built binaries at all, it only ships the source code. The binaries which can be found on the download page of the curl web and installed from other places on the Internet are all built and provided to the world by other friendly people and organizations.

The source code consists of a large number of files containing C code. Generally speaking, the same set of files are used to build binaries for all platforms and computer architectures that curl supports. curl can be built and run on a vast number of different platforms. If you use a rare operating system yourself, chances are that building curl from source is the easiest or perhaps the only way to get curl.

Making it easy to build curl is a priority to the curl project, although we don't always necessarily succeed!

git vs tarballs

When release tarballs are created, a few files are generated and included in the final release file. Those generated files are not present in the git repository, because they are generated and there is no need to store them in git.

So, if you want to build curl from git you need to generate some of those files yourself before you can build. On Linux and Unix systems, do this by running `./buildconf` and on Windows you run `buildconf.bat`.

On Linux and Unix-like systems

There are two distinctly different ways to build curl on Linux and other Unix-like systems. There's the one using the configure script and there's the CMake approach.

There are two different build environments to cater for people's different opinions and tastes. The configure based build is arguably the more mature and more complete build system and should probably be considered the default one.

Autotools

The "Autotools" is a collection of different tools that used together generate the `configure` script. The configure script is run by the user who wants to build curl and it does a whole bunch of things:

- it checks for features and functions present in your system
- it offers command-line options so that you as a builder can decide what to enable and disable in the build. Features and protocols, etc., can be toggled on/off. Or even compiler warning levels and more.
- it offers command-line options to let the builder point to specific installation paths for various third-party dependencies that curl can be built to use.
- specifies on which file path the generated installation should be placed when ultimately the build is made and "make install" is invoked

In the most basic usage, just running `./configure` in the source directory is enough. When the script completes, it outputs a summary of what options it has detected/enabled and what features that are still disabled, some of them possibly because it failed to detect the presence of necessary third-party dependencies that are needed for those functions to work.

Then you invoke `make` to build the entire thing and `make install` to install curl, libcurl and associated things. `make install` requires that you have the correct rights in your system to create and write files in the installation directory or you will get some errors.

cross-compiling

Cross-compiling means that you build the source on one architecture but the output is created to be run on a different one. For example, you could build the source on a Linux machine but have the output work to run on a Windows machine.

For cross-compiling to work, you need a dedicated compiler and build system setup for the particular target system for which you want to build. How to get and install that system is not covered in this book.

Once you have a cross compiler, you can instruct configure to use that compiler instead of the "native" compiler when it builds curl so that the end result then can be moved over and used on the other machine.

CMake

TBD

static linking

TBD

On Windows

TBD

make

TBD

CMake

TBD

other compilers

TBD

On other systems

TBD

Porting curl to non-supported systems

TBD

Select TLS backend

The configure based build offers the user to select from a wide variety of different TLS libraries when building. You select them by using the correct command line options.

The default OpenSSL configure check will also detect and use BoringSSL or libressl.

- GnuTLS: `--without-ssl --with-gnutls`.
- Cyassl: `--without-ssl --with-cyassl`
- NSS: `--without-ssl --with-nss`
- PolarSSL: `--without-ssl --with-polarssl`

- mbedTLS: `--without-ssl --with-mbedtls`
- axTLS: `--without-ssl --with-axtls`
- schannel: `--without-ssl --with-winssl`
- secure transport: `--with-winssl --with-darwinssl`

Dependencies

A key to making good software is to build on top of other great software. By using libraries that many others use, we reinvent the same things fewer times and we get more reliable software as there are more people using the same code.

A whole slew of features that curl provides require that it is built to use one or more external libraries. They are then dependencies of curl. None of them are *required*, but most users will want to use at least some of them.

zlib

https://zlib.net/

curl can do automatic decompression of data transferred over HTTP if built with zlib. Getting compressed data over the wire will use less bandwidth.

c-ares

https://c-ares.haxx.se/

curl can be built with c-ares to be able to do asynchronous name resolution. Another option to enable asynchronous name resolution is to build curl with the threaded name resolver backend, which will then instead create a separate helper thread for each name resolve. c-ares does it all within the same thread.

libssh2

https://libssh2.org/

When curl is built with libssh2, it enables support for the SCP and SFTP protocols.

nghttp2

https://nghttp2.org/

This is a library for handling HTTP/2 framing and is a prerequisite for curl to support HTTP version 2.

openldap

https://www.openldap.org/

This library is one option to allow curl to get support for the LDAP and LDAPS URL schemes. On Windows, you can also opt to build curl to use the winssl library.

librtmp

https://rtmpdump.mplayerhq.hu/

To enable curl's support for the RTMP URL scheme, you must build curl with the librtmp library that comes from the RTMPDump project.

libmetalink

https://launchpad.net/libmetalink

Build curl with libmetalink to have it support the metalink format, which allows curl to download the same file from multiple places. It includes checksums and more. See curl's --metalink option.

libpsl

https://rockdaboot.github.io/libpsl/

When you build curl with support for libpsl, the cookie parser will know about the Public Suffix List and thus handle such cookies appropriately.

libidn2

https://www.gnu.org/software/libidn/libidn2/manual/libidn2.html

curl handles International Domain Names (IDN) with the help of the libidn2 library.

TLS libraries

There are many different TLS libraries to choose from, so they are covered in a separate section.

Build to use a TLS library

To make curl support TLS based protocols, such as HTTPS, FTPS, SMTPS, POP3S, IMAPS and more, you need to build with a third-party TLS library since curl doesn't implement the TLS protocol itself.

curl is written to work with a large number of TLS libraries:

- BoringSSL
- GSKit (OS/400 specific)
- GnuTLS
- NSS
- OpenSSL
- Secure Transport (native macOS)
- WolfSSL
- axTLS
- libressl
- mbedTLS
- Schannel (native Windows)

When you build curl and libcurl to use one of these libraries, it is important that you have the library and its include headers installed on your build machine.

configure

Below, you will learn how to tell configure to use the different libraries. Note that for all libraries except OpenSSL and its siblings, you must *disable* the check for OpenSSL by using `--without-ssl`.

OpenSSL, BoringSSL, libressl

```
./configure
```

configure will detect OpenSSL in its default path by default. You can optionally point configure to a custom install path prefix where it can find openssl:

```
./configure --with-ssl=/home/user/installed/openssl
```

The alternatives BoringSSL and libressl look similar enough that configure will detect them the same way as OpenSSL but it will use some additional measures to find out which of the particular flavors it is using.

GnuTLS

```
./configure --with-gnutls --without-ssl
```

configure will detect GnuTLS in its default path by default. You can optionally point configure to a custom install path prefix where it can find gnutls:

```
./configure --with-gnutls=/home/user/installed/gnutls --without-ssl
```

NSS

```
./configure --with-nss --without-ssl
```

configure will detect NSS in its default path by default. You can optionally point configure to a custom install path prefix where it can find nss:

```
./configure --with-nss=/home/user/installed/nss --without-ssl
```

WolfSSL

```
./configure --with-cyassl --without-ssl
```

(cyassl was the former name of the library) configure will detect WolfSSL in its default path by default. You can optionally point configure to a custom install path prefix where it can find WolfSSL:

```
./configure --with-cyassl=/home/user/installed/nss --without-ssl
```

axTLS

```
./configure --with-axtls --without-ssl
```

configure will detect axTLS in its default path by default. You can optionally point configure to a custom install path prefix where it can find axTLS:

```
./configure --with-axtls=/home/user/installed/axtls --without-ssl
```

mbedTLS

```
./configure --with-mbedtls --without-ssl
```

configure will detect mbedTLS in its default path by default. You can optionally point configure to a custom install path prefix where it can find mbedTLS:

```
./configure --with-mbedtls=/home/user/installed/mbedtls --without-ssl
```

Secure Transport

```
./configure --with-darwinssl --without-ssl
```

(DarwinSSL is an alternative name for Secure Transport) configure will detect Secure Transport in its default path by default. You can optionally point configure to a custom install path prefix where it can find DarwinSSL:

```
./configure --with-darwinssl=/home/user/installed/darwinssl --without-ssl
```

Schannel

```
./configure --with-winssl --without-ssl
```

(WinSSL is an alternative name for Schannel) configure will detect Schannel in its default path by default. You can optionally point configure to a custom install path prefix where it can find WinSSL:

```
./configure --with-winssl=/home/user/installed/winssl --without-ssl
```

Build curl with boringssl

build boringssl

$HOME/src is where I put the code in this example. You can pick wherever you like.

```
$ cd $HOME/src
$ git clone https://boringssl.googlesource.com/boringssl
$ cd boringssl
$ mkdir build
$ cd build
$ cmake ..
$ make
```

set up the build tree to get detected by curl's configure

In the boringssl source tree root, make sure there's a `lib` and an `include` dir. The `lib` dir should contain the two libs (I made them symlinks into the build dir). The `include` dir is already present by default. Make and populate `lib` like this (commands issued in the source tree root, not in the `build/` subdirectory).

```
$ mkdir lib
$ cd lib
$ ln -s ../build/ssl/libssl.a
$ ln -s ../build/crypoto/libcrypto.a
```

configure curl

`LIBS=-lpthread ./configure --with-ssl=$HOME/src/boringssl` (where I point out the root of the boringssl tree)

Verify that at the end of the configuration, it says it detected BoringSSL to be used.

build curl

Run `make` in the curl source tree.

Now you can install curl normally with `make install` etc.

libcurl basics

The engine in the curl command-line tool is libcurl. libcurl is also the engine in thousands of tools, services and applications out there today, performing their Internet data transfers.

C API

libcurl is a library of functions that are provided with a C API, for applications written in C. You can easily use it from C++ too, with only a few considerations (see libcurl for C++ programmers. For other languages, there exist "bindings" that work as intermediate layers between libcurl the library and corresponding functions for the particular language you like.

Transfer oriented

We have designed libcurl to be transfer oriented usually without forcing users to be protocol experts or in fact know much at all about networking or the protocols involved. You setup a transfer with as many details and specific information as you can and want, and then you tell libcurl to perform that transfer.

That said, networking and protocols are areas with lots of pitfalls and special cases so the more you know about these things, the more you will be able to understand about libcurl's options and ways of working. Not to mention, such knowledge is invaluable when you are debugging and need to understand what to do next when things don't go as you intended.

The most basic libcurl using application can be as small as just a couple of lines of code, but most applications will, of course, need more code than that.

Simple by default, more on demand

libcurl generally does the simple and basic transfer by default, and if you want to add more advanced features, you add that by setting the correct options. For example, libcurl doesn't support HTTP cookies by default but it does once you tell it.

This makes libcurl's behaviors easier to guess and depend on, and also it makes it easier to maintain old behavior and add new features. Only applications that actually ask for and use the new features will get that behavior.

Easy handle

The fundamentals you need to learn with libcurl:

First you create an "easy handle", which is your handle to a transfer, really:

```
CURL *easy_handle = curl_easy_init();
```

Then you set various options in that handle to control the upcoming transfer. Like, this example sets the URL:

```
/* set URL to operate on */
res = curl_easy_setopt(easy_handle, CURLOPT_URL, "http://example.com/");
```

Creating the easy handle and setting options on it doesn't make any transfer happen, and usually don't even make much more happen other than libcurl storing your wish to be used later when the transfer actually occurs. Lots of syntax checking and validation of the input may also be postponed, so just because `curl_easy_setopt` didn't complain, it doesn't mean that the input was correct and valid; you may get an error returned later.

Read more on easy options in its separate section.

All options are "sticky". They remain set in the handle until you change them again, or call `curl_easy_reset()` on the handle.

When you are done setting options to your easy handle, you can fire off the actual transfer.

The actual "perform the transfer phase" can be done using different means and function calls, depending on what kind of behavior you want in your application and how libcurl is best integrated into your architecture. Those are further described later in this chapter.

After the transfer has completed, you can figure out if it succeeded or not and you can extract stats and various information that libcurl gathered during the transfer from the easy handle. See Post transfer information.

While the transfer is ongoing, libcurl calls your specified functions—known as *callbacks*—to deliver data, to read data or to do a wide variety of things.

Reuse!

Easy handles are meant and designed to be reused. When you have done a single transfer with the easy handle, you can immediately use it again for your next transfer. There are lots of gains to be had by this.

"Drive" transfers

libcurl provides three different ways to perform the transfer. Which way to use in your case is entirely up to you and what you need.

1. The 'easy' interface lets you do a single transfer in a synchronous fashion. libcurl will do the entire transfer and return control back to your application when it is completed—successful or failed.
2. The 'multi' interface is for when you want to do more than one transfer at the same time, or you just want a non-blocking transfer.
3. The 'multi_socket' interface is a slight variation of the regular multi one, but is event-based and is really the suggested API to use if you intend to scale up the number of simultaneous transfers to hundreds or thousands or so.

Let's look at each one a little closer…

Driving with the easy interface

The name 'easy' was picked simply because this is really the easy way to use libcurl, and with easy, of course, comes a few limitations. Like, for example, that it can only do one transfer at a time and that it does the entire transfer in a single function call and returns once it is completed:

```
res = curl_easy_perform( easy_handle );
```

If the server is slow, if the transfer is large or if you have some unpleasant timeouts in the network or similar, this function call can end up taking a very long time. You can, of course, set timeouts to not allow it to spend more than N seconds, but it could still mean a substantial amount of time depending on the particular conditions.

If you want your application to do something else while libcurl is transferring with the easy interface, you need to use multiple threads. If you want to do multiple simultaneous transfers when using the easy interface, you need to perform each of the transfers in its own thread.

Driving with the multi interface

The name 'multi' is for multiple, as in multiple parallel transfers, all done in the same single thread. The multi API is non-blocking so it can also make sense to use it for single transfers.

The transfer is still set in an "easy" `CURL *` handle as described above, but with the multi interface you also need a multi `CURLM *` handle created and use that to drive all the individual transfers. The multi handle can "hold" one or many easy handles:

```
CURLM *multi_handle = curl_multi_init();
```

A multi handle can also get certain options set, which you do with `curl_multi_setopt()`, but in the simplest case you might not have anything to set there.

To drive a multi interface transfer, you first need to add all the individual easy handles that should be transferred to the multi handle. You can add them to the multi handle at any point and you can remove them again whenever you like. Removing an easy handle from a multi handle will, of course, remove the association and that particular transfer would stop immediately.

Adding an easy handle to the multi handle is very easy:

```
curl_multi_add_handle( multi_handle, easy_handle );
```

Removing one is just as easily done:

```
curl_multi_remove_handle( multi_handle, easy_handle );
```

Having added the easy handles representing the transfers you want to perform, you write the transfer loop. With the multi interface, you do the looping so you can ask libcurl for a set of file descriptors and a timeout value and do the `select()` call yourself, or you can use the slightly simplified version which does that for us, with `curl_multi_wait`. The simplest loop would basically be this: (*note that a real application would check return codes*)

```
int transfers_running;
do {
   curl_multi_wait ( multi_handle, NULL, 0, 1000, NULL);
   curl_multi_perform ( multi_handle, &transfers_running );
} while (transfers_running);
```

The fourth argument to `curl_multi_wait`, set to 1000 in the example above, is a timeout in milliseconds. It is the longest time the function will wait for any activity before it returns anyway. You don't want to lock up for too long before calling `curl_multi_perform` again as there are timeouts, progress callbacks and more that may loose precision if you do so.

To instead do select() on our own, we extract the file descriptors and timeout value from libcurl like this (*note that a real application would check return codes*):

```
int transfers_running;
do {
  fd_set fdread;
  fd_set fdwrite;
  fd_set fdexcep;
  int maxfd = -1;
  long timeout;

  /* extract timeout value */
  curl_multi_timeout(multi_handle, &timeout);
  if (timeout < 0)
    timeout = 1000;

  /* convert to struct usable by select */
  timeout.tv_sec = timeout / 1000;
  timeout.tv_usec = (timeout % 1000) * 1000;

  FD_ZERO(&fdread);
  FD_ZERO(&fdwrite);
  FD_ZERO(&fdexcep);

  /* get file descriptors from the transfers */
  mc = curl_multi_fdset(multi_handle, &fdread, &fdwrite, &fdexcep, &maxfd);

  if (maxfd == -1) {
    SHORT_SLEEP;
  }
  else
   select(maxfd+1, &fdread, &fdwrite, &fdexcep, &timeout);

  /* timeout or readable/writable sockets */
  curl_multi_perform(multi_handle, &transfers_running);
} while ( transfers_running );
```

Both these loops let you use one or more file descriptors of your own on which to wait, like if you read from your own sockets or a pipe or similar.

And again, you can add and remove easy handles to the multi handle at any point during the looping. Removing a handle mid-transfer will, of course, abort that transfer.

When is a single transfer done?

As the examples above show, a program can detect when an individual transfer completes by seeing that the `transfers_running` variable decreases.

It can also call `curl_multi_info_read()`, which will return a pointer to a struct (a "message") if a transfer has ended and you can then find out the result of that transfer using that struct.

When you do multiple parallel transfers, more than one transfer can of course complete in the same `curl_multi_perform` invocation and then you might need more than one call to `curl_multi_info_read` to get info about each completed transfer.

Driving with the "multi_socket" interface

multi_socket is the extra spicy version of the regular multi interface and is designed for event-driven applications. Make sure you read the Drive with multi interface section first.

multi_socket supports multiple parallel transfers—all done in the same single thread—and have been used to run several tens of thousands of transfers in a single application. It is usually the API that makes the most sense if you do a large number (>100 or so) of parallel transfers.

Event-driven in this case means that your application uses a system level library or setup that "subscribes" to a number of sockets and it lets your application know when one of those sockets are readable or writable and it tells you exactly which one.

This setup allows clients to scale up the number of simultaneous transfers much higher than with other systems, and still maintain good performance. The "regular" APIs otherwise waste far too much time scanning through lists of all the sockets.

Pick one

There are numerous event based systems to select from out there, and libcurl is completely agnostic to which one you use. libevent, libev are libuv three popular ones but you can also go directly to your operating system's native solutions such as epoll, kqueue, /dev/poll, pollset, Event Completion or I/O Completion Ports.

Many easy handles

Just like with the regular multi interface, you add easy handles to a multi handle with `curl_multi_add_handle()`. One easy handle for each transfer you want to perform.

You can add them at any time while the transfers are running and you can also similarly remove easy handles at any time using the `curl_multi_remove_handle` call. Typically though, you remove a handle only after its transfer is completed.

multi_socket callbacks

As explained above, this event-based mechanism relies on the application to know which sockets are used by libcurl and what libcurl waits for on those sockets: if it waits for the socket to become readable, writable or both!

It also needs to tell libcurl when its timeout time has expired, as it is control of driving everything libcurl can't do it itself. So libcurl must tell the application an updated timeout value, too.

socket_callback

libcurl informs the application about socket activity to wait for with a callback called CURLMOPT_SOCKETFUNCTION. Your application needs to implement such a function:

```
int socket_callback(CURL *easy,      /* easy handle */
                    curl_socket_t s, /* socket */
                    int what,        /* what to wait for */
                    void *userp,     /* private callback pointer */
                    void *socketp)   /* private socket pointer */
{
   /* told about the socket 's' */
}

/* set the callback in the multi handle */
curl_multi_setopt(multi_handle, CURLMOPT_SOCKETFUNCTION, socket_callback);
```

Using this, libcurl will set and remove sockets your application should monitor. Your application tells the underlying event-based system to wait for the sockets. This callback will be called multiple times if there are multiple sockets to wait for, and it will be called again when the status changes and perhaps you should switch from waiting for a writable socket to instead wait for it to become readable.

When one of the sockets that the application is monitoring on libcurl's behalf registers that it becomes readable or writable, as requested, you tell libcurl about it by calling `curl_multi_socket_action()` and passing in the affected socket and an associated bitmask specifying which socket activity that was registered:

```
int running_handles;
ret = curl_multi_socket_action(multi_handle,
                               sockfd, /* the socket with activity */
                               ev_bitmask, /* the specific activity */
                               &running_handles);
```

timer_callback

The application is in control and will wait for socket activity. But even without socket activity there will be things libcurl needs to do. Timeout things, calling the progress callback, starting over a retry or failing a transfer that takes too long, etc. To make that work, the application must also make sure to handle a single-shot timeout that libcurl sets.

libcurl sets the timeout with the timer_callback CURLMOPT_TIMERFUNCTION:

```
int timer_callback(multi_handle,   /* multi handle */
                   timeout_ms,     /* milliseconds to wait */
                   userp)          /* private callback pointer */
{
  /* the new time-out value to wait for is in 'timeout_ms' */
}

/* set the callback in the multi handle */
curl_multi_setopt(multi_handle, CURLMOPT_TIMERFUNCTION, timer_callback);
```

There is only one timeout for the application to handle for the entire multi handle, no matter how many individual easy handles that have been added or transfers that are in progress. The timer callback will be updated with the current nearest-in-time period to wait. If libcurl gets called before the timeout expiry time because of socket activity, it may very well update the timeout value again before it expires.

When the event system of your choice eventually tells you that the timer has expired, you need to tell libcurl about it:

```
curl_multi_socket_action(multi, CURL_SOCKET_TIMEOUT, 0, &running);
```

…in many cases, this will make libcurl call the timer_callback again and set a new timeout for the next expiry period.

How to start everything

When you have added one or more easy handles to the multi handle and set the socket and timer callbacks in the multi handle, you are ready to start the transfer.

To kick it all off, you tell libcurl it timed out (because all easy handles start out with a very, very short timeout) which will make libcurl call the callbacks to set things up and from then on you can can just let your event system drive:

```
/* all easy handles and callbacks are setup */

curl_multi_socket_action(multi, CURL_SOCKET_TIMEOUT, 0, &running);

/* now the callbacks should have been called and we have sockets to wait for
   and possibly a timeout, too. Make the event system do its magic */

event_base_dispatch(event_base); /* libevent2 has this API */

/* at this point we have exited the event loop */
```

When is it done?

The 'running_handles' counter returned by `curl_multi_socket_action` holds the number of current transfers not completed. When that number reaches zero, we know there are no transfers going on.

Each time the 'running_handles' counter changes, `curl_multi_info_read()` will return info about the specific transfers that completed.

Connection reuse

libcurl keeps a pool of old connections alive. When one transfer has completed it will keep N connections alive in a "connection pool" (sometimes also called connection cache) so that a subsequent transfer that happens to be able to reuse one of the existing connections can use it instead of creating a new one. Reusing a connection instead of creating a new one offers significant benefits in speed and required resources.

When libcurl is about to make a new connection for the purposes of doing a transfer, it will first check to see if there's an existing connection in the pool that it can reuse instead. The connection re-use check is done before any DNS or other name resolving mechanism is used, so it is purely host name based. If there's an existing live connection to the right host name, a lot of other properties (port number, protocol, etc) are also checked to see that it can be used.

Easy API pool

When you are using the easy API, or, more specifically, `curl_easy_perform()`, libcurl will keep the pool associated with the specific easy handle. Then reusing the same easy handle will ensure it can reuse its connection.

Multi API pool

When you are using the multi API, the connection pool is instead kept associated with the multi handle. This allows you to cleanup and re-create easy handles freely without risking losing the connection pool, and it allows the connection used by one easy handle to get reused by a separate one in a later transfer. Just reuse the multi handle!

Sharing the "connection cache"

Since libcurl 7.57.0, applications can use the share interface to have otherwise independent transfers share the same connection pool.

Callbacks

Lots of operations within libcurl are controlled with the use of *callbacks*. A callback is a function pointer provided to libcurl that libcurl then calls at some point in time to get a particular job done.

Each callback has its specific documented purpose and it requires that you write it with the exact function prototype to accept the correct arguments and return the documented return code and return value so that libcurl will perform the way you want it to.

Each callback option also has a companion option that sets the associated "user pointer". This user pointer is a pointer that libcurl doesn't touch or care about, but just passes on as an argument to the callback. This allows you to, for example, pass in pointers to local data all the way through to your callback function.

Unless explicitly stated in a libcurl function documentation, it is not legal to invoke libcurl functions from within a libcurl callback.

Write callback

The write callback is set with `CURLOPT_WRITEFUNCTION`:

```
curl_easy_setopt(handle, CURLOPT_WRITEFUNCTION, write_callback);
```

The `write_callback` function must match this prototype:

```
size_t write_callback(char *ptr, size_t size, size_t nmemb, void *userdata);
```

This callback function gets called by libcurl as soon as there is data received that needs to be saved. *ptr* points to the delivered data, and the size of that data is *size* multiplied with *nmemb*.

If this callback isn't set, libcurl instead uses 'fwrite' by default.

The write callback will be passed as much data as possible in all invokes, but it must not make any assumptions. It may be one byte, it may be thousands. The maximum amount of body data that will be passed to the write callback is defined in the curl.h header file: `CURL_MAX_WRITE_SIZE` (the usual default is 16KB). If `CURLOPT_HEADER` is enabled for this transfer, which makes header data get passed to the write callback, you can get up to `CURL_MAX_HTTP_HEADER` bytes of header data passed into it. This usually means 100KB.

This function may be called with zero bytes data if the transferred file is empty.

The data passed to this function will not be zero terminated! You cannot, for example, use printf's "%s" operator to display the contents nor strcpy to copy it.

This callback should return the number of bytes actually taken care of. If that number differs from the number passed to your callback function, it will signal an error condition to the library. This will cause the transfer to get aborted and the libcurl function used will return `CURLE_WRITE_ERROR`.

The user pointer passed in to the callback in the *userdata* argument is set with `CURLOPT_WRITEDATA`:

```
curl_easy_setopt(handle, CURLOPT_WRITEDATA, custom_pointer);
```

Read callback

The read callback is set with `CURLOPT_READFUNCTION`:

```
curl_easy_setopt(handle, CURLOPT_READFUNCTION, read_callback);
```

The `read_callback` function must match this prototype:

```
size_t read_callback(char *buffer, size_t size, size_t nitems, void *stream);
```

This callback function gets called by libcurl when it wants to send data to the server. This is a transfer that you have set up to upload data or otherwise send it off to the server. This callback will be called over and over until all data has been delivered or the transfer failed.

The **stream** pointer points to the private data set with `CURLOPT_READDATA`:

```
curl_easy_setopt(handle, CURLOPT_READDATA, custom_pointer);
```

If this callback isn't set, libcurl instead uses 'fread' by default.

The data area pointed at by the pointer **buffer** should be filled up with at most **size** multiplied with **nitems** number of bytes by your function. The callback should then return the number of bytes that it stored in that memory area, or 0 if we have reached the end of the data. The callback can also return a few "magic" return codes to cause libcurl to return failure immediately or to pause the particular transfer. See the CURLOPT_READFUNCTION man page for details.

Progress callback

The progress callback is what gets called regularly and repeatedly for each transfer during the entire lifetime of the transfer. The old callback was set with `CURLOPT_PROGRESSFUNCTION` but the modern and preferred callback is set with `CURLOPT_XFERINFOFUNCTION` :

```
curl_easy_setopt(handle, CURLOPT_XFERINFOFUNCTION, xfer_callback);
```

The `xfer_callback` function must match this prototype:

```
int xfer_callback(void *clientp, curl_off_t dltotal, curl_off_t dlnow,
                  curl_off_t ultotal, curl_off_t ulnow);
```

If this option is set and `CURLOPT_NOPROGRESS` is set to 0 (zero), this callback function gets called by libcurl with a frequent interval. While data is being transferred it will be called very frequently, and during slow periods like when nothing is being transferred it can slow down to about one call per second.

The **clientp** pointer points to the private data set with `CURLOPT_XFERINFODATA` :

```
curl_easy_setopt(handle, CURLOPT_XFERINFODATA, custom_pointer);
```

The callback gets told how much data libcurl will transfer and has transferred, in number of bytes:

- **dltotal** is the total number of bytes libcurl expects to download in this transfer.
- **dlnow** is the number of bytes downloaded so far.
- **ultotal** is the total number of bytes libcurl expects to upload in this transfer.
- **ulnow** is the number of bytes uploaded so far.

Unknown/unused argument values passed to the callback will be set to zero (like if you only download data, the upload size will remain 0). Many times the callback will be called one or more times first, before it knows the data sizes, so a program must be made to handle that.

Returning a non-zero value from this callback will cause libcurl to abort the transfer and return `CURLE_ABORTED_BY_CALLBACK` .

If you transfer data with the multi interface, this function will not be called during periods of idleness unless you call the appropriate libcurl function that performs transfers.

(The deprecated callback `CURLOPT_PROGRESSFUNCTION` worked identically but instead of taking arguments of type `curl_off_t` , it used `double` .)

Header callback

The header callback is set with `CURLOPT_HEADERFUNCTION` :

```
curl_easy_setopt(handle, CURLOPT_HEADERFUNCTION, header_callback);
```

The `header_callback` function must match this prototype:

```
size_t header_callback(char *ptr, size_t size, size_t nmemb, void *userdata);
```

This callback function gets called by libcurl as soon as a header has been received. *ptr* points to the delivered data, and the size of that data is *size* multiplied with *nmemb*. libcurl buffers headers and delivers only "full" headers, one by one, to this callback.

The data passed to this function will not be zero terminated! You cannot, for example, use printf's "%s" operator to display the contents nor strcpy to copy it.

This callback should return the number of bytes actually taken care of. If that number differs from the number passed to your callback function, it signals an error condition to the library. This will cause the transfer to abort and the libcurl function used will return `CURLE_WRITE_ERROR` .

The user pointer passed in to the callback in the *userdata* argument is set with `CURLOPT_HEADERDATA` :

```
curl_easy_setopt(handle, CURLOPT_HEADERDATA, custom_pointer);
```

Debug callback

The debug callback is set with `CURLOPT_DEBUGFUNCTION`:

```
curl_easy_setopt(handle, CURLOPT_DEBUGFUNCTION, debug_callback);
```

The `debug_callback` function must match this prototype:

```
int debug_callback(CURL *handle,
                   curl_infotype type,
                   char *data,
                   size_t size,
                   void *userdata);
```

This callback function replaces the default verbose output function in the library and will get called for all debug and trace messages to aid applications to understand what's going on. The *type* argument explains what sort of data that is provided: header, data or SSL data and in which direction it flows.

A common use for this callback is to get a full trace of all data that libcurl sends and receives. The data sent to this callback is always the unencrypted version, even when, for example, HTTPS or other encrypted protocols are used.

This callback must return zero or cause the transfer to stop with an error code.

The user pointer passed in to the callback in the *userdata* argument is set with `CURLOPT_DEBUGDATA`:

```
curl_easy_setopt(handle, CURLOPT_DEBUGDATA, custom_pointer);
```

sockopt callback

The sockopt callback is set with `CURLOPT_SOCKOPTFUNCTION`:

```
curl_easy_setopt(handle, CURLOPT_SOCKOPTFUNCTION, sockopt_callback);
```

The `sockopt_callback` function must match this prototype:

```
int sockopt_callback(void *clientp,
                     curl_socket_t curlfd,
                     curlsocktype purpose);
```

This callback function gets called by libcurl when a new socket has been created but before the connect call, to allow applications to change specific socket options.

The **clientp** pointer points to the private data set with `CURLOPT_SOCKOPTDATA`:

```
curl_easy_setopt(handle, CURLOPT_SOCKOPTDATA, custom_pointer);
```

This callback should return:

- CURL_SOCKOPT_OK on success
- CURL_SOCKOPT_ERROR to signal an unrecoverable error to libcurl
- CURL_SOCKOPT_ALREADY_CONNECTED to signal success but also that the socket is in fact already connected to the destination

SSL context callback

TBD

seek and ioctl callbacks

TBD

Convert to and from network callbacks

TBD

Convert from UTF-8 callback

TBD

Opensocket and closesocket callbacks

Occasionally you end up in a situation where you want your application to control with more precision exactly what socket libcurl will use for its operations. libcurl offers this pair of callbacks that replaces libcurl's own call to `socket()` and the subsequent `close()` of the same file descriptor.

Provide a file descriptor

By setting the `CURLOPT_OPENSOCKETFUNCTION` callback, you can provide a custom function to return a file descriptor for libcurl to use:

```
curl_easy_setopt(handle, CURLOPT_OPENSOCKETFUNCTION, opensocket_callback);
```

The `opensocket_callback` function must match this prototype:

```
curl_socket_t opensocket_callback(void *clientp,
                                  curlsocktype purpose,
                                  struct curl_sockaddr *address);
```

The callback gets the *clientp* as first argument, which is simply an opaque pointer you set with `CURLOPT_OPENSOCKETDATA`.

The other two arguments pass in data that identifies for what *purpose* and *address* the socket is to be used. The *purpose* is a typedef with a value of `CURLSOCKTYPE_IPCXN` or `CURLSOCKTYPE_ACCEPT`, basically identifying in which circumstance the socket is created. The "accept" case being when libcurl is used to accept an incoming FTP connection for when FTP active mode is used, and all other cases when libcurl creates a socket for its own outgoing connections the *IPCXN* value is passed in.

The *address* pointer points to a `struct curl_sockaddr` that describes the IP address of the network destination for which this socket is created. Your callback can for example use this information to whitelist or blacklist specific addresses or address ranges.

The socketopen callback is also explicitly allowed to modify the target address in that struct, if you would like to offer some sort of network filter or translation layer.

The callback should return a file descriptor or `CURL_SOCKET_BAD`, which then will cause an unrecoverable error within libcurl and it will eventually return `CURLE_COULDNT_CONNECT` from its perform function.

If you want to return a file descriptor that is *already connected* to a server, then you must also set the sockopt callback and make sure that returns the correct return value.

The `curl_sockaddress` struct looks like this:

```
struct curl_sockaddr {
  int family;
  int socktype;
  int protocol;
  unsigned int addrlen;
  struct sockaddr addr;
};
```

Socket close callback

The corresponding callback to the open socket is of course the close socket. Usually when you provide a custom way to provide a file descriptor you want to provide your own cleanup version as well:

```
curl_easy_setopt(handle, CURLOPT_CLOSEOCKETFUNCTION, closesocket_callback);
```

The `closesocket_callback` function must match this prototype:

```
int closesocket_callback(void *clientp, curl_socket_t item);
```

SSH key callback

TBD

RTSP interleave callback

TBD

FTP chunk callbacks

TBD

FTP matching callback

TBD

Cleanup

In previous sections we have discussed how to setup handles and how to drive the transfers. All transfers will, of course, end up at some point, either successfully or with a failure.

Multi API

When you have finished a single transfer with the multi API, you use `curl_multi_info_read()` to identify exactly which easy handle was completed and you remove that easy handle from the multi handle with `curl_multi_remove_handle()`.

If you remove the last easy handle from the multi handle so there are no more transfers going on, you can close the multi handle like this:

```
curl_multi_cleanup( multi_handle );
```

easy handle

When the easy handle is done serving its purpose, you can close it. If you intend to do another transfer, you are however advised to rather reuse the handle rather than to close it and create a new one.

If you don't intend to do another transfer with the easy handle, you simply ask libcurl to cleanup:

```
curl_easy_cleanup( easy_handle );
```

Name resolving

Most transfers libcurl can do involves a name that first needs to be translated to an internet address. That's "name resolving". Using a numerical IP address directly in the URL usually avoids the name resolve phase, but in many cases it isn't easy to manually replace the name with the IP address.

libcurl tries very hard to re-use an existing connection rather than to create and connection a new one. The function that checks for an existing connection to use is based purely on the name and is performed before any name resolving is attempted. That's one of the reasons the re-use is so much faster. A transfer using a reused connection will not resolve the host name again.

If no connection can be reused, libcurl resolves the host name to the set of addresses it resolves to. Typically this means asking for both IPv4 and IPv6 addresses and there may be a whole set of those returned to libcurl. That set of addresses is then tried until one works, or it returns failure.

An application can force libcurl to use only an IPv4 or IPv6 resolved address by setting `CURLOPT_IPRESOLVE` to the preferred value. For example, ask to only use IPv6 addresses:

```
curl_easy_setopt(easy, CURLOPT_IPRESOLVE, CURL_IPRESOLVE_V6);
```

Name resolver backends

libcurl can be built to do name resolves in at least three different ways and depending on which backend way that was used, it gets a slightly different feature set and sometimes modified behavior.

1. The default backend is invoking the "normal" libc resolver functions in a new helper-thread, so that it can still do fine-grained timeouts if wanted and there will be no blocking calls involved.

2. On older systems, libcurl uses the standard synchronous name resolver functions. They unfortunately make all transfers within a multi handle block during its operation and it is much harder to time out nicely.

3. There's also support for resolving with the c-ares third party library, which supports asynchronous name resolving without the use of threads. This scales better to huge number of parallel transfers but it isn't always 100% compatible with the native name

resolver functionality.

Caching

When a name has been resolved, the result will be put in libcurl's in-memory cache so that subsequent resolves of the same name will be near instant for as long the name is kept in the DNS cache. By default, each entry is kept in the cache for 60 seconds, but that value can be changed with `CURLOPT_DNS_CACHE_TIMEOUT`.

The DNS cache is kept within the easy handle when `curl_easy_perform` is used, or within the multi handle when the multi interface is used. It can also be made shared between multiple easy handles using the share interface.

Custom addresses for hosts

Sometimes it is handy to provide "fake" addresses to real host names so that libcurl will connect to a different address instead of one an actual name resolve would suggest.

With the help of the CURLOPT_RESOLVE option, an application can pre-populate libcurl's DNS cache with a custom address for a given host name and port number.

To make libcurl connect to 127.0.0.1 when example.com on port 443 is requested, an application can do:

```
struct curl_slist *dns;
dns = curl_slist_append(NULL, "example.com:443:127.0.0.1");
curl_easy_setopt(curl, CURLOPT_RESOLVE, dns);
```

Since this puts the "fake" address into the DNS cache, it will work even when following redirects etc.

Name server options

For libcurl built to use c-ares, there's a few options available that offer fine-grained control of what DNS servers to use and how. This is limited to c-ares build purely because these are powers that are not available when the standard system calls for name name resolving are used.

- With `CURLOPT_DNS_SERVERS`, the application can select to use a set of dedicated DNS servers.

- With `CURLOPT_DNS_INTERFACE` it can tell libcurl which network interface to speak DNS over instead of the default one.
- With `CURLOPT_DNS_LOCAL_IP4` and `CURLOPT_DNS_LOCAL_IP6`, the application can specify which specific network addresses to bind DNS resolves to.

Global DNS cache is bad

The is a *deprecated* option called `CURLOPT_DNS_USE_GLOBAL_CACHE` that when enabled tells curl to use a global DNS cache. This cache has no locks and stores data in a global context that then can be shared by all other easy handles that also is set to use the global cache.

This option should only be used by legacy applications and you *should* work on converting this over to using the share interface for sharing DNS cache the "proper" way. This option will be removed in a future version.

Proxies

A proxy in a network context is a sort of middle man, a server in between you as a client and the remote server you want to communicate with. The client contacts the middle man which then goes on to contact the remote server for you.

This sort of proxy use is sometimes used by companies and organizations, in which case you are usually required to use them to reach the target server.

There are several different kinds of proxies and different protocols to use when communicating with a proxy, and libcurl supports a few of the most common proxy protocols. It is important to realize that the protocol used to the proxy isn't necessarily the same protocol used to the remote server.

When setting up a transfer with libcurl you need to point out the server name and port number of the proxy. You may find that your favorite browsers can do this in slightly more advanced ways than libcurl can, and we will get into such details in later sections.

Proxy types

libcurl supports the two major proxy types: SOCKS and HTTP proxies. More specifically, it supports both SOCKS4 and SOCKS5 with or without remote name lookup, as well as both HTTP and HTTPS to the local proxy.

The easiest way to specify which kind of proxy you are talking to is to set the scheme part of the proxy host name string (`CURLOPT_PROXY`) to match it:

```
socks4://proxy.example.com:12345/
socks4a://proxy.example.com:12345/
socks5://proxy.example.com:12345/
socks5h://proxy.example.com:12345/
http://proxy.example.com:12345/
https://proxy.example.com:12345/
```

`socks4` - means SOCKS4 with local name resolving

`socks4a` - means SOCKS4 with proxy's name resolving

`socks5` - means SOCKS5 with local name resolving

`socks5h` - means SOCKS5 with proxy's name resolving

`http` - means HTTP, which always lets the proxy resolve names

`https` - means HTTPS **to the proxy**, which always lets the proxy resolve names (Note that HTTPS proxy support was added recently, in curl 7.52.0, and it still only works with a subset of the TLS libraries: OpenSSL, GnuTLS and NSS.)

You can also opt to set the type of the proxy with a separate option if you prefer to only set the host name, using `CURLOPT_PROXYTYPE`. Similarly, you can set the proxy port number to use with `CURLOPT_PROXYPORT`.

Local or proxy name lookup

In a section above you can see that different proxy setups allow the name resolving to be done by different parties involved in the transfer. You can in several cases either have the client resolve the server host name and pass on the IP address to the proxy to connect to - which of course assumes that the name lookup works accurately on the client system - or you can hand over the name to the proxy to have the proxy resolve the name; converting it to an IP address to connect to.

When you are using an HTTP or HTTPS proxy, you always give the name to the proxy to resolve.

Which proxy?

TBD

Using proxies for various protocols

TBD

HTTP proxy

TBD

HTTPS proxy

TBD

Proxy authentication

TBD

Post transfer info

Remember how libcurl transfers are associated with an "easy handle"! Each transfer has such a handle and when a transfer is completed, before the handle is cleaned or reused for another transfer, it can be used to extract information from the previous operation.

Your friend for doing this is called `curl_easy_getinfo()` and you tell it which specific information you're interested in and it will return that to you if it can.

When you use this function, you pass in the easy handle, which information you want and a pointer to a variable to hold the answer. You must pass in a pointer to a variable of the correct type or you risk that things will go side-ways. These information values are designed to be provided *after* the transfer is completed.

The data you receive can be a long, a 'char *', a 'struct curl_slist* ', a double or a socket.

This is how you extract the `Content-Type:` value from the previous HTTP transfer:

```
CURLcode res;
char *content_type;
res = curl_easy_getinfo(curl, CURLINFO_CONTENT_TYPE, &content_type);
```

but if you want to extract the local port number that was used in that connection:

```
CURLcode res;
long port_number;
res = curl_easy_getinfo(curl, CURLINFO_LOCAL_PORT, &port_number);
```

Available information

Getinfo option	Type	Description
CURLINFO_ACTIVESOCKET	curl_socket_t	The session's active
CURLINFO_APPCONNECT_TIME	double	Time from start until SSL/SSH handshake completed.
CURLINFO_CERTINFO	struct curl_slist *	Certificate chain
CURLINFO_CONDITION_UNMET	long	Whether or not a time conditional was met

CURLINFO_CONNECT_TIME	double	Time from start until r host or proxy comple
CURLINFO_CONTENT_LENGTH_DOWNLOAD	double	Content length from t Content-Length heac
CURLINFO_CONTENT_LENGTH_UPLOAD	double	Upload size
CURLINFO_CONTENT_TYPE	char *	Content type from the Content-Type header
CURLINFO_COOKIELIST	struct curl_slist *	List of all known cook
CURLINFO_EFFECTIVE_URL	char *	Last used URL
CURLINFO_FILETIME	long	Remote time of the re document
CURLINFO_FTP_ENTRY_PATH	char *	The entry path after l in to an FTP server
CURLINFO_HEADER_SIZE	long	Number of bytes of a headers received
CURLINFO_HTTPAUTH_AVAIL	long	Available HTTP authentication metho (bitmask)
CURLINFO_HTTP_CONNECTCODE	long	Last proxy CONNEC response code
CURLINFO_HTTP_VERSION	long	The http version usec connection
CURLINFO_LASTSOCKET	long	Last socket used
CURLINFO_LOCAL_IP	char *	Local-end IP address connection
CURLINFO_LOCAL_PORT	long	Local-end port of last connection
CURLINFO_NAMELOOKUP_TIME	double	Time from start until r resolving completed
CURLINFO_NUM_CONNECTS	long	Number of new succe connections used for previous transfer
CURLINFO_OS_ERRNO	long	The errno from the la failure to connect
CURLINFO_PRETRANSFER_TIME	double	Time from start until j before the transfer be
CURLINFO_PRIMARY_IP	char *	IP address of the last connection

CURLINFO_PRIMARY_PORT	long	Port of the last conne
CURLINFO_PRIVATE	char *	User's private data p
CURLINFO_PROTOCOL	long	The protocol used fo connection
CURLINFO_PROXYAUTH_AVAIL	long	Available HTTP prox authentication metho
CURLINFO_PROXY_SSL_VERIFYRESULT	long	Proxy certificate veri result
CURLINFO_REDIRECT_COUNT	long	Total number of redir that were followed
CURLINFO_REDIRECT_TIME	double	Time taken for all red steps before the final transfer
CURLINFO_REDIRECT_URL	char *	URL a redirect would you to, had you enab redirects
CURLINFO_REQUEST_SIZE	long	Number of bytes sen issued HTTP reques
CURLINFO_RESPONSE_CODE	long	Last received respon code
CURLINFO_RTSP_CLIENT_CSEQ	long	RTSP CSeq that will used
CURLINFO_RTSP_CSEQ_RECV	long	RTSP CSeq last rece
CURLINFO_RTSP_SERVER_CSEQ	long	RTSP CSeq that will expected
CURLINFO_RTSP_SESSION_ID	char *	RTSP session ID
CURLINFO_SCHEME	char *	The scheme used fo connection
CURLINFO_SIZE_DOWNLOAD	double	Number of bytes downloaded
CURLINFO_SIZE_UPLOAD	double	Number of bytes uplo
CURLINFO_SPEED_DOWNLOAD	double	Average download s
CURLINFO_SPEED_UPLOAD	double	Average upload spee
CURLINFO_SSL_ENGINES	struct curl_slist *	A list of OpenSSL cr engines
CURLINFO_SSL_VERIFYRESULT	long	Certificate verificatio
CURLINFO_STARTTRANSFER_TIME	double	Time from start until when the first byte is received

CURLINFO_TLS_SESSION	struct curl_slist *	TLS session info that used for further proce (**Deprecated option, CURLINFO_TLS_SS instead!**)
CURLINFO_TLS_SSL_PTR	struct curl_slist *	TLS session info that used for further proce
CURLINFO_TOTAL_TIME	double	Total time of previous transfer

Share data between handles

Sometimes applications need to share data between transfers. All easy handles added to the same multi handle automatically get a lot of sharing done between the handles in that same multi handle, but sometimes that's not exactly what you want.

Multi handle

All easy handles added to the same multi handle automatically share cookies, connection cache, dns cache and SSL session id cache.

Sharing between easy handles

libcurl has a generic "sharing interface", where the application creates a "share object" that then holds data that can be shared by any number of easy handles. The data is then stored and read from the shared object instead of kept within the handles that are sharing it.

```
CURLSH *share = curl_share_init();
```

The shared object can be set to share all or any of cookies, connection cache, dns cache and SSL session id cache.

For example, setting up the share to hold cookies and dns cache:

```
curl_share_setopt(share, CURLSHOPT_SHARE, CURL_LOCK_DATA_COOKIE);
curl_share_setopt(share, CURLSHOPT_SHARE, CURL_LOCK_DATA_DNS);
```

... and then you setup the corresponding transfer to use this share object:

```
curl_easy_setopt(curl, CURLOPT_SHARE, share);
```

Transfers done with this `curl` handle will thus use and store its cookie and dns information in the `share` handle. You can set several easy handles to share the same share object.

What to share

`CURL_LOCK_DATA_COOKIE` - set this bit to share cookie jar. Note that each easy handle still needs to get its cookie "engine" started properly to start using cookies.

`CURL_LOCK_DATA_DNS` - the DNS cache is where libcurl stores addresses for resolved host names for a while to make subsequent lookups faster.

`CURL_LOCK_DATA_SSL_SESSION` - the SSL session ID cache is where libcurl store resume information for SSL connections to be able to resume a previous connection faster.

`CURL_LOCK_DATA_CONNECT` - when set, this handle will use a shared connection cache and thus will probably be more likely to find existing connections to re-use etc, which may result in faster performance when doing multiple transfers to the same host in a serial manner.

Locking

If you want have the share object shared by transfers in a multi-threaded environment. Perhaps you have a CPU with many cores and you want each core to run its own thread and transfer data, but you still want the different transfers to share data. Then you need to set the mutex callbacks.

If you don't use threading and you *know* you access the shared object in a serial one-at-a-time manner you don't need to set any locks. But if there is ever more than one transfer that access share object at a time, it needs to get mutex callbacks setup to prevent data destruction and possibly even crashes.

Since libcurl itself doesn't know how to lock things or even what threading model you're using, you must make sure to do mutex locks that only allows one access at a time. A lock callback for a pthreads-using application could look similar to:

```
static void lock_cb(CURL *handle, curl_lock_data data,
                    curl_lock_access access, void *userptr)
{
  pthread_mutex_lock(&lock[data]); /* uses a global lock array */
}
curl_share_setopt(share, CURLSHOPT_LOCKFUNC, lock_cb);
```

With the corresponding unlock callback could look like:

```
static void unlock_cb(CURL *handle, curl_lock_data data,
                      void *userptr)
{
  pthread_mutex_unlock(&lock[data]); /* uses a global lock array */
}
curl_share_setopt(share, CURLSHOPT_UNLOCKFUNC, unlock_cb);
```

Unshare

TBD

API compatibility

libcurl promises API stability and guarantees that your program written today will remain working in the future. We don't break compatibility.

Over time, we add features, new options and new functions to the APIs but we do not change behavior in a non-compatible way or remove functions.

The last time we changed the API in an non-compatible way was for 7.16.0 in 2006 and we plan to never do it again.

Version numbers

Curl and libcurl are individually versioned, but they mostly follow each other rather closely.

The version numbering is always built up using the same system:

```
X.Y.Z
```

- X is main version number
- Y is release number
- Z is patch number

Bumping numbers

One of these X.Y.Z numbers will get bumped in every new release. The numbers to the right of a bumped number will be reset to zero.

The main version number X is bumped when *really* big, world colliding changes are made. The release number Y is bumped when changes are performed or things/features are added. The patch number Z is bumped when the changes are mere bugfixes.

It means that after a release 1.2.3, we can release 2.0.0 if something really big has been made, 1.3.0 if not that big changes were made or 1.2.4 if mostly bugs were fixed.

Bumping, as in increasing the number with 1, is unconditionally only affecting one of the numbers (and the ones to the right of it are set to zero). 1 becomes 2, 3 becomes 4, 9 becomes 10, 88 becomes 89 and 99 becomes 100. So, after 1.2.9 comes 1.2.10. After 3.99.3, 3.100.0 might come.

All original curl source release archives are named according to the libcurl version (not according to the curl client version that, as said before, might differ).

Which libcurl version

As a service to any application that might want to support new libcurl features while still being able to build with older versions, all releases have the libcurl version stored in the `curl/curlver.h` file using a static numbering scheme that can be used for comparison. The version number is defined as:

```
#define LIBCURL_VERSION_NUM 0xXXYYZZ
```

Where XX, YY and ZZ are the main version, release and patch numbers in hexadecimal. All three number fields are always represented using two digits (eight bits each). 1.2.0 would appear as "0x010200" while version 9.11.7 appears as "0x090b07".

This 6-digit hexadecimal number is always a greater number in a more recent release. It makes comparisons with greater than and less than work.

This number is also available as three separate defines: `LIBCURL_VERSION_MAJOR`, `LIBCURL_VERSION_MINOR` and `LIBCURL_VERSION_PATCH`.

These defines are, of course, only suitable to figure out the version number built *just now* and they won't help you figuring out which libcurl version that is used at run-time three years from now.

Which libcurl version runs

To figure out which libcurl version that your application is using *right now*, `curl_version_info()` is there for you.

Applications should use this function to judge if things are possible to do or not, instead of using compile-time checks, as dynamic/DLL libraries can be changed independent of applications.

curl_version_info() returns a pointer to a struct with information about version numbers and various features and in the running version of libcurl. You call it by giving it a special age counter so that libcurl knows the "age" of the libcurl that calls it. The age is a define called `CURLVERSION_NOW` and is a counter that is increased at irregular intervals throughout the curl development. The age number tells libcurl what struct set it can return.

You call the function like this:

curl_version_info_data *ver = curl_version_info(CURLVERSION_NOW);

The data will then be pointing at struct that has or at least can have the following layout:

```
struct {
  CURLversion age;          /* see description below */

  /* when 'age' is 0 or higher, the members below also exist: */
  const char *version;      /* human readable string */
  unsigned int version_num; /* numeric representation */
  const char *host;         /* human readable string */
  int features;             /* bitmask, see below */
  char *ssl_version;        /* human readable string */
  long ssl_version_num;     /* not used, always zero */
  const char *libz_version; /* human readable string */
  const char * const *protocols; /* protocols */

  /* when 'age' is 1 or higher, the members below also exist: */
  const char *ares;         /* human readable string */
  int ares_num;             /* number */

  /* when 'age' is 2 or higher, the member below also exists: */
  const char *libidn;       /* human readable string */

  /* when 'age' is 3 or higher (7.16.1 or later), the members below also
     exist  */
  int iconv_ver_num;       /* '_libiconv_version' if iconv support enabled */

  const char *libssh_version; /* human readable string */

} curl_version_info_data;
```

curl --libcurl

We actively encourage users to first try out the transfer they want to do with the curl command-line tool, and once it works roughly the way you want it to, you append the `--libcurl [filename]` option to the command line and run it again.

The `--libcurl` command-line option will create a C program in the provided file name. That C program is an application that uses libcurl to run the transfer you just had the curl command-line tool do. There are some exceptions and it isn't always a 100% match, but you will find that it can serve as an excellent inspiration source for what libcurl options you want or can use and what additional arguments to provide to them.

If you specify the filename as a single dash, as in `--libcurl -` you will get the program written to stdout instead of a file.

As an example, we run a command to just get http://example.com:

```
curl http://example.com --libcurl example.c
```

This creates `example.c` in the current directory, looking similar to this:

```
/********* Sample code generated by the curl command-line tool **********
 * All curl_easy_setopt() options are documented at:
 * https://curl.haxx.se/libcurl/c/curl_easy_setopt.html
 ************************************************************************/
#include <curl/curl.h>

int main(int argc, char *argv[])
{
  CURLcode ret;
  CURL *hnd;

  hnd = curl_easy_init();
  curl_easy_setopt(hnd, CURLOPT_URL, "http://example.com");
  curl_easy_setopt(hnd, CURLOPT_NOPROGRESS, 1L);
  curl_easy_setopt(hnd, CURLOPT_USERAGENT, "curl/7.45.0");
  curl_easy_setopt(hnd, CURLOPT_MAXREDIRS, 50L);
  curl_easy_setopt(hnd, CURLOPT_SSH_KNOWNHOSTS, "/home/daniel/.ssh/known_hosts");
  curl_easy_setopt(hnd, CURLOPT_TCP_KEEPALIVE, 1L);

  /* Here is a list of options the curl code used that cannot get generated
     as source easily. You may select to either not use them or implement
     them yourself.

  CURLOPT_WRITEDATA set to a objectpointer
  CURLOPT_WRITEFUNCTION set to a functionpointer
  CURLOPT_READDATA set to a objectpointer
  CURLOPT_READFUNCTION set to a functionpointer
  CURLOPT_SEEKDATA set to a objectpointer
  CURLOPT_SEEKFUNCTION set to a functionpointer
  CURLOPT_ERRORBUFFER set to a objectpointer
  CURLOPT_STDERR set to a objectpointer
  CURLOPT_HEADERFUNCTION set to a functionpointer
  CURLOPT_HEADERDATA set to a objectpointer

  */

  ret = curl_easy_perform(hnd);

  curl_easy_cleanup(hnd);
  hnd = NULL;

  return (int)ret;
}
/**** End of sample code ****/
```

Header files

There is only ever one header your libcurl using application needs to include:

```
#include <curl/curl.h>
```

That file in turn includes a few other public header files but you can basically pretend they don't exist. (Historically speaking, we started out slightly different but over time we have stabilized around this form of only using a single one for includes.)

Global initialization

Before you do anything libcurl related in your program, you should do a global libcurl initialize call with `curl_global_init()`. This is necessary because some underlying libraries that libcurl might be using need a call ahead to get setup and initialized properly.

curl_global_init() is, unfortunately, not thread safe, so you must ensure that you only do it once and never simultaneously with another call. It initializes global state so you should only call it once, and once your program is completely done using libcurl you can call `curl_global_cleanup()` to free and clean up the associated global resources the init call allocated.

libcurl is built to handle the situation where you skip the `curl_global_init()` call, but it does so by calling it itself instead (if you didn't do it before any actual file transfer starts) and it then uses its own defaults. But beware that it is still not thread safe even then, so it might cause some "interesting" side effects for you. It is much better to call curl_global_init() yourself in a controlled manner.

libcurl multi-threading

libcurl is thread safe but has no internal thread synchronization. You may have to provide your own locking or change options to properly use libcurl threaded. Exactly what is required depends on how libcurl was built. Please refer to the libcurl thread safety webpage, which contains the latest information.

Set handle options

You set options in the easy handle to control how that transfer is going to be done, or in some cases you can actually set options and modify the transfer's behavior while it is in progress. You set options with `curl_easy_setopt()` and you provide the handle, the option you want to set and the argument to the option. All options take exactly one argument and you must always pass exactly three parameters to the curl_easy_setopt() calls.

Since the curl_easy_setopt() call accepts several hundred different options and the various options accept a variety of different types of arguments, it is very important to read up on the specifics and provide exactly the argument type the specific option supports and expects. Passing in the wrong type can lead to unexpected side-effects or hard to understand hiccups.

The perhaps most important option that every transfer needs, is the URL. libcurl cannot perform a transfer without knowing which URL it concerns so you must tell it. The URL option name is `CURLOPT_URL` as all options are prefixed with `CURLOPT_` and then the descriptive name—all using uppercase letters. An example line setting the URL to get the "http://example.com" HTTP contents could look like:

```
CURLcode ret = curl_easy_setopt(easy, CURLOPT_URL, "http://example.com");
```

Again: this only sets the option in the handle. It will not do the actual transfer or anything. It will basically just tell libcurl to copy the string and if that works it returns OK.

It is, of course, good form to check the return code to see that nothing went wrong.

Setting numerical options

Since curl_easy_setopt() is a vararg function where the 3rd argument can use different types depending on the situation, normal C language type conversion cannot be done. So you **must** make sure that you truly pass a 'long' and not an 'int' if the documentation tells you so. On architectures where they are the same size, you may not get any problems but not all work like that. Similarly, for options that accept a 'curl_off_t' type, it is **crucial** that you pass in an argument using that type and no other.

Enforce a long:

```
curl_easy_setopt(handle, CURLOPT_TIMEOUT, 5L); /* 5 seconds timeout */
```

Enforce a curl_off_t:

```
curl_off_t no_larger_than = 0x50000;
curl_easy_setopt(handle, CURLOPT_MAXFILE_LARGE, no_larger_than);
```

Get handle options

No, there's no general method to extract the same information you previously set with `curl_easy_setopt()`! If you need to be able to extract the information again that you set earlier, then we encourage you to keep track of that data yourself in your application.

CURLcode return code

Many libcurl functions return a CURLcode. That's a special libcurl typedefed variable for error codes. It returns `CURLE_OK` (which has the value zero) if everything is fine and dandy and it returns a non-zero number if a problem was detected. There are almost one hundred `CURLcode` errors in use, and you can find them all in the `curl/curl.h` header file and documented in the libcurl-errors man page.

You can convert a CURLcode into a human readable string with the `curl_easy_strerror()` function—but be aware that these errors are rarely phrased in a way that is suitable for anyone to expose in a UI or to an end user:

```
const char *str = curl_easy_strerror( error );
printf("libcurl said %s\n", str);
```

Another way to get a slightly better error text in case of errors is to set the `CURLOPT_ERRORBUFFER` option to point out a buffer in your program and then libcurl will store a related error message there before it returns an error:

```
curl error[CURL_ERROR_SIZE]; /* needs to be at least this big */
CURLcode ret = curl_easy_setopt(handle, CURLOPT_ERRORBUFFER, error);
```

Verbose operations

Okay, we just showed how to get the error as a human readable text as that is an excellent help to figure out what went wrong in a particular transfer and often explains why it can be done like that or what the problem is for the moment.

The next lifesaver when writing libcurl applications that everyone needs to know about and needs to use extensively, at least while developing libcurl applications or debugging libcurl itself, is to enable "verbose mode" with `CURLOPT_VERBOSE`:

```
CURLcode ret = curl_easy_setopt(handle, CURLOPT_VERBOSE, 1L);
```

When libcurl is told to be verbose it will mention transfer-related details and information to stderr while the transfer is ongoing. This is awesome to figure out why things fail and to learn exactly what libcurl does when you ask it different things. You can redirect the output elsewhere by changing stderr with `CURLOPT_STDERR` or you can get even more info in a fancier way with the debug callback (explained further in a later section).

Trace everything

Verbose is certainly fine, but sometimes you need more. libcurl also offers a trace callback that in addition to showing you all the stuff the verbose mode does, it also passes on *all* data sent and received so that your application gets a full trace of everything.

The sent and received data passed to the trace callback is given to the callback in its unencrypted form, which can be very handy when working with TLS or SSH based protocols when capturing the data off the network for debugging isn't very practical.

When you set the `CURLOPT_DEBUGFUNCTION` option, you still need to have `CURLOPT_VERBOSE` enabled but with the trace callback set libcurl will use that callback instead of its internal handling.

The trace callback should match a prototype like this:

```
int my_trace(CURL *handle, curl_infotype type, char *ptr, size_t size,
             void *userp);
```

handle is the easy handle it concerns, **type** describes the particular data passed to the callback (data in/out, header in/out, TLS data in/out and "text"), **ptr** points to the data being **size** number of bytes. **userp** is the custom pointer you set with `CURLOPT_DEBUGDATA`.

The data pointed to by **ptr** *will not* be zero terminated, but will be exactly of the size as told by the **size** argument.

The callback must return 0 or libcurl will consider it an error and abort the transfer.

On the curl web site, we host an example called debug.c that includes a simple trace function to get inspiration from.

There are also additional details in the CURLOPT_DEBUGFUNCTION man page.

libcurl examples

The native API for libcurl is in C so this chapter is focussed on examples written in C. But since many language bindings for libcurl are thin, they usually expose more or less the same functions and thus they can still be interesting and educational for users of other languages, too.

Get a simple HTML page

This example just fetches the HTML and sends it to stdout:

```
#include <stdio.h>
#include <curl/curl.h>

int main(void)
{
  CURL *curl;
  CURLcode res;

  curl = curl_easy_init();
  if(curl) {
    curl_easy_setopt(curl, CURLOPT_URL, "http://example.com/");

    /* Perform the request, res will get the return code */
    res = curl_easy_perform(curl);
    /* Check for errors */
    if(res != CURLE_OK)
      fprintf(stderr, "curl_easy_perform() failed: %s\n",
              curl_easy_strerror(res));

    /* always cleanup */
    curl_easy_cleanup(curl);
  }
  return 0;
}
```

Get a HTML page in memory

This example is a variation of the former that instead of sending the data to stdout (which often is not what you want), this stores the received data in a memory buffer that is made larger as the incoming data grows.

It accomplishes this by using the write callback to receive the data.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <curl/curl.h>

struct MemoryStruct {
  char *memory;
  size_t size;
};

static size_t
WriteMemoryCallback(void *contents, size_t size, size_t nmemb, void *userp)
{
  size_t realsize = size * nmemb;
  struct MemoryStruct *mem = (struct MemoryStruct *)userp;

  mem->memory = realloc(mem->memory, mem->size + realsize + 1);
  if(mem->memory == NULL) {
    /* out of memory! */
    printf("not enough memory (realloc returned NULL)\n");
    return 0;
  }

  memcpy(&(mem->memory[mem->size]), contents, realsize);
  mem->size += realsize;
  mem->memory[mem->size] = 0;

  return realsize;
}

int main(void)
{
  CURL *curl_handle;
  CURLcode res;

  struct MemoryStruct chunk;

  chunk.memory = malloc(1);  /* will be grown as needed by the realloc above */
  chunk.size = 0;    /* no data at this point */

  curl_global_init(CURL_GLOBAL_ALL);

  /* init the curl session */
  curl_handle = curl_easy_init();

  /* specify URL to get */
  curl_easy_setopt(curl_handle, CURLOPT_URL, "http://www.example.com/");

  /* send all data to this function  */
```

```
  curl_easy_setopt(curl_handle, CURLOPT_WRITEFUNCTION, WriteMemoryCallback);

  /* we pass our 'chunk' struct to the callback function */
  curl_easy_setopt(curl_handle, CURLOPT_WRITEDATA, (void *)&chunk);

  /* some servers don't like requests that are made without a user-agent
     field, so we provide one */
  curl_easy_setopt(curl_handle, CURLOPT_USERAGENT, "libcurl-agent/1.0");

  /* get it! */
  res = curl_easy_perform(curl_handle);

  /* check for errors */
  if(res != CURLE_OK) {
    fprintf(stderr, "curl_easy_perform() failed: %s\n",
            curl_easy_strerror(res));
  }
  else {
    /*
     * Now, our chunk.memory points to a memory block that is chunk.size
     * bytes big and contains the remote file.
     *
     * Do something nice with it!
     */

    printf("%lu bytes retrieved\n", (long)chunk.size);
  }

  /* cleanup curl stuff */
  curl_easy_cleanup(curl_handle);

  free(chunk.memory);

  /* we're done with libcurl, so clean it up */
  curl_global_cleanup();

  return 0;
}
```

Submit a login form over HTTP

TBD

Get an FTP directory listing

TBD

Download an HTTPS page straight into memory

TBD

Upload data to an HTTP site without blocking

TBD

libcurl for C++ programmers

TBD

- strings are C strings, not C++ string objects
- callback considerations

HTTP with libcurl

HTTP is by far the most commonly used protocol by libcurl users and libcurl offers countless ways of modifying such transfers. See the HTTP protocol basics for some basics on how the HTTP protocol works.

HTTPS

TBD

HTTP proxy

TBD

HTTP responses

Every HTTP request includes a HTTP response. A HTTP response is a set of metadata and a response body, where the body can occasionally be zero bytes and thus nonexistent. A HTTP response will however always have response headers.

The response body will be passed to the write callback and the response headers to the header callback, but sometimes an application just want to know the size of the data.

The size of a response *as told by the server headers* can be extracted with `curl_easy_getinfo()` like this:

```
double size;
curl_easy_getinfo(curl, CURLINFO_CONTENT_LENGTH_DOWNLOAD, &size);
```

but if you can wait until after the transfer is already done, which also is a more reliable way since not all URLs will provide the size up front (like for example for servers that generate content on demand) you can instead ask for the amount of downloaded data in the most recent transfer.

```
double size;
curl_easy_getinfo(curl, CURLINFO_SIZE_DOWNLOAD, &size);
```

HTTP response code

Every HTTP response starts off with a single line that contains the HTTP response code. It is a three digit number that contains the server's idea of the status for the request. The numbers are detailed in the HTTP standard specifications but they are divided into ranges that basically work like this:

Code	Meaning
1xx	Transient code, a new one follows
2xx	Things are OK
3xx	The content is somewhere else
4xx	Failed because of a client problem
5xx	Failed because of a server problem

You can extract the response code after a transfer like this

```
long code;
curl_easy_getinfo(curl, CURLINFO_RESPONSE_CODE, &code);
```

About HTTP response code "errors"

While the response code numbers can include numbers (in the 4xx and 5xx ranges) which the server uses to signal that there was an error processing the request, it is important to realize that this will not cause libcurl to return an error.

When libcurl is asked to perform a HTTP transfer it will return an error if that HTTP transfer fails. However, getting a HTTP 404 or the like back is not a problem for libcurl. It is not a HTTP transfer error. A user might very well be writing a client for testing a server's HTTP responses.

If you insist on curl treating HTTP response codes from 400 and up as errors, libcurl offers the `CURLOPT_FAILONERROR` option that if set instructs curl to return `CURLE_HTTP_RETURNED_ERROR` in this case. It will then return error as soon as possible and not deliver the response body.

HTTP Requests

A HTTP request is what curl sends to the server when it tells the server what to do. When it wants to get data or send data. All transfers involving HTTP starts with a HTTP request.

A HTTP request contains a method, a path, HTTP version and a set of request headers. And of course a libcurl using application can tweak all those fields.

Request method

Every HTTP request contains a "method", sometimes referred to as a "verb". It is usually something like GET, HEAD, POST or PUT but there are also more esoteric ones like DELETE, PATCH and OPTIONS.

Usually when you use libcurl to set up and perform a transfer the specific request method is implied by the options you use. If you just ask for a URL, it means the method will be `GET` while if you set for example `CURLOPT_POSTFIELDS` that will make libcurl use the `POST` method. If you set `CURLOPT_UPLOAD` to true, libcurl will send a `PUT` method in its HTTP request and so on. Asking for `CURLOPT_NOBODY` will make libcurl use `HEAD`.

However, sometimes those default HTTP methods are not good enough or simply not the ones you want your transfer to use. Then you can instruct libcurl to use the specific method you like with `CURLOPT_CUSTOMREQUEST`. For example, you want to send a `DELETE` method to the URL of your choice:

```
curl_easy_setopt(curl, CURLOPT_CUSTOMREQUEST, "DELETE");
curl_easy_setopt(curl, CURLOPT_URL, "https://example.com/file.txt");
```

The CURLOPT_CUSTOMREQUEST setting should only be the single keyword to use as method in the HTTP request line. If you want to change or add additional HTTP request headers, see the following section.

Customize HTTP request headers

When libcurl issues HTTP requests as part of performing the data transfers you've asked it to, it will of course send them off with a set of HTTP headers that are suitable for fulfilling the task given to it.

If just given the URL "http://localhost/file1.txt", libcurl 7.51.0 would send the following request to the server:

```
GET /file1.txt HTTP/1.1
Host: localhost
Accept: */*
```

If you would instead instruct your application to also set `CURLOPT_POSTFIELDS` to the string "foobar" (6 letters, the quotes only used for visual delimiters here), it would send the following headers:

```
POST /file1.txt HTTP/1.1
Host: localhost
Accept: */*
Content-Length: 6
Content-Type: application/x-www-form-urlencoded
```

If you're not pleased with the default set of headers libcurl sends, the application has the power to add, change or remove headers in the HTTP request.

Add a header

To add a header that wouldn't otherwise be in the request, add it with `CURLOPT_HTTPHEADER`. Suppose you want a header called `Name:` that contains `Mr. Smith`:

```
struct curl_slist *list = NULL;
list = curl_slist_append(list, "Name: Mr Smith");
curl_easy_setopt(curl, CURLOPT_HTTPHEADER, list);
curl_easy_perform(curl);
curl_slist_free_all(list); /* free the list again */
```

Change a header

If one of those default headers aren't to your satisfaction you can alter them. Like if you think the default `Host:` header is wrong (even though it is derived from the URL you give libcurl), you can tell libcurl your own:

```
struct curl_slist *list = NULL;
list = curl_slist_append(list, "Host: Alternative");
curl_easy_setopt(curl, CURLOPT_HTTPHEADER, list);
curl_easy_perform(curl);
curl_slist_free_all(list); /* free the list again */
```

Remove a header

When you think libcurl uses a header in a request that you really think it shouldn't, you can easily tell it to just remove it from the request. Like if you want to take away the `Accept:` header. Just provide the header name with nothing to the right sight of the colon:

```
struct curl_slist *list = NULL;
list = curl_slist_append(list, "Accept:");
curl_easy_setopt(curl, CURLOPT_HTTPHEADER, list);
curl_easy_perform(curl);
curl_slist_free_all(list); /* free the list again */
```

Provide a header without contents

As you may then have noticed in the above sections, if you try to add a header with no contents on the right side of the colon, it will be treated as a removal instruction and it will instead completely inhibit that header from being sent. If you instead *truly* want to send a header with zero contents on the right side, you need to use a special marker. You must provide the header with a semicolon instead of a proper colon. Like `Header;`. So if you want to add a header to the outgoing HTTP request that is just `Moo:` with nothing following the colon, you could write it like:

```
struct curl_slist *list = NULL;
list = curl_slist_append(list, "Moo;");
curl_easy_setopt(curl, CURLOPT_HTTPHEADER, list);
curl_easy_perform(curl);
curl_slist_free_all(list); /* free the list again */
```

Referrer

The `Referer:` header (yes, it is misspelled) is a standard HTTP header that tells the server from which URL the user-agent was directed from when it arrived at the URL it now requests. It is a normal header so you can set it yourself with the `CURLOPT_HEADER` approach as shown above, or you can use the shortcut known as `CURLOPT_REFERER`. Like this:

```
curl_easy_setopt(curl, CURLOPT_REFERER, "https://example.com/fromhere/");
curl_easy_perform(curl);
```

Automatic referrer

When libcurl is asked to follow redirects itself with the `CURLOPT_FOLLOWLOCATION` option, and you still want to have the `Referer:` header set to the correct previous URL from where it did the redirect, you can ask libcurl to set that by itself:

```
curl_easy_setopt(curl, CURLOPT_FOLLOWLOCATION, 1L);
curl_easy_setopt(curl, CURLOPT_AUTOREFERER, 1L);
curl_easy_setopt(curl, CURLOPT_URL, "https://example.com/redirected.cgi");
curl_easy_perform(curl);
```

HTTP versions

As any other Internet protocol, the HTTP protocol has kept evolving over the years and now there are clients and servers distributed over the world and over time that speak different versions with varying levels of success. So in order to get libcurl to work with the URLs you pass in libcurl offers ways for you to specify which HTTP version that request and transfer should use. libcurl is designed in a way so that it tries to use the most common, the most sensible if you want, default values first but sometimes that isn't enough and then you may need to instruct libcurl what to do.

Since mid 2016, libcurl defaults to use HTTP/2 for HTTPS servers if you have a libcurl that has HTTP/2 abilities built-in, libcurl will attempt to use HTTP/2 automatically or fall down to 1.1 in case the negotiation failed. Non-HTTP/2 capable libcurls get 1.1 over HTTPS by default. Plain HTTP requests still default to HTTP/1.1.

If the default behavior isn't good enough for your transfer, the `CURLOPT_HTTP_VERSION` option is there for you.

Option	Description
CURL_HTTP_VERSION_NONE	Reset back to default behavior
CURL_HTTP_VERSION_1_0	Enforce use of the legacy HTTP/1.0 protocol version
CURL_HTTP_VERSION_1_1	Do the request using the HTTP/1.1 protocol version
CURL_HTTP_VERSION_2_0	Attempt to use HTTP/2
CURL_HTTP_VERSION_2TLS	Attempt to use HTTP/2 on HTTPS connections only, otherwise do HTTP/1.1
CURL_HTTP_VERSION_2_PRIOR_KNOWLEDGE	Use HTTP/2 straight away without "upgrading" from 1.1. It requires that you know that this server is OK with it.

HTTP ranges

What if the client only wants the first 200 bytes out of a remote resource or perhaps 300 bytes somewhere in the middle? The HTTP protocol allows a client to ask for only a specific data range. The client asks the server for the specific range with a start offset and an end offset. It can even combine things and ask for several ranges in the same request by just listing a bunch of pieces next to each other. When a server sends back multiple independent pieces to answer such a request, you will get them separated with mime boundary strings and it will be up to the user application to handle that accordingly. curl will not further separate such a response.

However, a byte range is only a request to the server. It does not have to respect the request and in many cases, like when the server automatically generates the contents on the fly when it is being asked, it will simply refuse to do it and it then instead respond with the full contents anyway.

You can make libcurl ask for a range with `CURLOPT_RANGE`. Like if you want the first 200 bytes out of something:

```
curl_easy_setopt(curl, CURLOPT_RANGE, "0-199");
```

Or everything in the file starting from index 200:

```
curl_easy_setopt(curl, CURLOPT_RANGE, "200-");
```

Get 200 bytes from index 0 *and* 200 bytes from index 1000:

```
curl_easy_setopt(curl, CURLOPT_RANGE, "0-199,1000-199");
```

Cookies with libcurl

By default and by design, libcurl makes transfers as basic as possible and features need to be enabled to get used. One such feature is HTTP cookies, more known as just plain and simply "cookies".

Cookies are name/value pairs sent by the server (using a `Set-Cookie:` header) to be stored in the client, and are then supposed to get sent back again in requests that matches the host and path requirements that were specified along with the cookie when it came from the server (using the `Cookie:` header). On the modern web of today, sites are known to sometimes use very large numbers of cookies.

Cookie engine

When you enable the "cookie engine" for a specific easy handle, it means that it will record incoming cookies, store them in the in-memory "cookie store" that is associated with the easy handle and subsequently send the proper ones back if an HTTP request is made that matches.

There are two ways to switch on the cookie engine:

Enable cookie engine with reading

Ask libcurl to import cookies into the easy handle from a given file name with the `CURLOPT_COOKIEFILE` option:

```
curl_easy_setopt(easy, CURLOPT_COOKIEFILE, "cookies.txt");
```

A common trick is to just specify a non-existing file name or plain "" to have it just activate the cookie engine with a blank cookie store to start with.

This option can be set multiple times and then each of the given files will be read.

Enable cookie engine with writing

Ask for received cookies to get stored in a file with the `CURLOPT_COOKIEJAR` option:

```
curl_easy_setopt(easy, CURLOPT_COOKIEJAR, "cookies.txt");
```

when the easy handle is closed later with `curl_easy_cleanup()`, all known cookies will be written to the given file. The file format is the well-known "Netscape cookie file" format that browsers also once used.

Setting custom cookies

A simpler and more direct way to just pass on a set of specific cookies in a request that doesn't add any cookies to the cookie store and doesn't even activate the cookie engine, is to set the set with `CURLOPT_COOKIE:':

```
curl_easy_setopt(easy, CURLOPT_COOKIE, "name=daniel; present=yes;");
```

The string you set there is the raw string that would be sent in the HTTP request and should be in the format of repeated sequences of `NAME=VALUE;` - including the semicolon separator.

Import export

The cookie in-memory store can hold a bunch of cookies, and libcurl offers very powerful ways for an application to play with them. You can set new cookies, you can replace an existing cookie and you can extract existing cookies.

Add a cookie to the cookie store

Add a new cookie to the cookie store by simply passing it into curl with `CURLOPT_COOKIELIST` with a new cookie. The format of the input is a single line in the cookie file format, or formatted as a `Set-Cookie:` response header, but we recommend the cookie file style:

```
#define SEP  "\\t"  /* Tab separates the fields */

char *my_cookie =
  "example.com"    /* Hostname */
  SEP "FALSE"      /* Include subdomains */
  SEP "/"          /* Path */
  SEP "FALSE"      /* Secure */
  SEP "0"          /* Expiry in epoch time format. 0 == Session */
  SEP "foo"        /* Name */
  SEP "bar";       /* Value */

curl_easy_setopt(curl, CURLOPT_COOKIELIST, my_cookie);
```

If that given cookie would match an already existing cookie (with the same domain and path, etc.), it would overwrite the old one with the new contents.

Get all cookies from the cookie store

Sometimes writing the cookie file when you close the handle isn't enough and then your application can opt to extract all the currently known cookies from the store like this:

```
struct curl_slist *cookies
curl_easy_getinfo(easy, CURLINFO_COOKIELIST, &cookies);
```

This returns a pointer to a linked list of cookies, and each cookie is (again) specified as a single line of the cookie file format. The list is allocated for you, so do not forget to call `curl_slist_free_all` when the application is done with the information.

Cookie store commands

If setting and extracting cookies isn't enough, you can also interfere with the cookie store in more ways:

Wipe the entire in-memory storage clean with:

```
curl_easy_setopt(curl, CURLOPT_COOKIELIST, "ALL");
```

Erase all session cookies (cookies without expiry date) from memory:

```
curl_easy_setopt(curl, CURLOPT_COOKIELIST, "SESS");
```

Force a write of all cookies to the file name previously specified with `CURLOPT_COOKIEJAR`:

```
curl_easy_setopt(curl, CURLOPT_COOKIELIST, "FLUSH");
```

Force a reload of cookies from the file name previously specified with `CURLOPT_COOKIEFILE`:

```
curl_easy_setopt(curl, CURLOPT_COOKIELIST, "RELOAD");
```

Cookie file format

The cookie file format is text based and stores one cookie per line. Lines that start with `#` are treated as comments.

Each line that each specifies a single cookie consists of seven text fields separated with TAB characters.

Field	Example	Meaning
0	example.com	Domain name
1	FALSE	Include subdomains boolean
2	/foobar/	Path
3	FALSE	Set over a secure transport
4	1462299217	Expires at – seconds since Jan 1st 1970, or 0
5	person	Name of the cookie
6	daniel	Value of the cookie

libcurl HTTP download

The GET method is the default method libcurl uses when a HTTP URL is requested and no particular other method is asked for. It asks the server for a particular resource—the standard HTTP download request:

```
easy = curl_easy_init();
curl_easy_setopt(easy, CURLOPT_URL, "http://example.com/");
curl_easy_perform(easy);
```

Since options set in an easy handle are sticky and remain until changed, there may be times when you have asked for another request method than GET and then want to switch back to GET again for a subsequent request. For this purpose, there's the `CURLOPT_HTTPGET` option:

```
curl_easy_setopt(easy, CURLOPT_HTTPGET, 1L);
```

Download headers too

A HTTP transfer also includes a set of response headers. Response headers are metadata associated with the actual payload, called the response body. All downloads will get a set of headers too, but when using libcurl you can select whether you want to have them downloaded (seen) or not.

You can ask libcurl to pass on the headers to the same "stream" as the regular body is, by using `CURLOPT_HEADER` :

```
easy = curl_easy_init();
curl_easy_setopt(easy, CURLOPT_HEADER, 1L);
curl_easy_setopt(easy, CURLOPT_URL, "http://example.com/");
curl_easy_perform(easy);
```

Or you can opt to store the headers in a separate download file, by relying on the default behaviors of the write and header callbacks:

```
easy = curl_easy_init();
FILE *file = fopen("headers", "wb");
curl_easy_setopt(easy, CURLOPT_HEADERDATA, file);
curl_easy_setopt(easy, CURLOPT_URL, "http://example.com/");
curl_easy_perform(easy);
fclose(file);
```

If you only want to casually browse the headers, you may even be happy enough with just setting verbose mode while developing as that will show both outgoing and incoming headers sent to stderr:

```
curl_easy_setopt(easy, CURLOPT_VERBOSE, 1L);
```

HTTP upload

Uploads over HTTP can be done in many different ways and it is important to notice the differences. They can use different methods, like POST or PUT, and when using POST the body formatting can differ.

In addition to those HTTP differences, libcurl offers different ways to provide the data to upload.

HTTP POST

POST is typically the HTTP method to pass data to a remote web application. A very common way to do that in browsers is by filling in a HTML form and pressing submit. It is the standard way for a HTTP request to pass on data to the server. With libcurl you normally provide that data as a pointer and a length:

```
curl_easy_setopt(easy, CURLOPT_POSTFIELDS, dataptr);
curl_easy_setopt(easy, CURLOPT_POSTFIELDSIZE, (long)datalength);
```

Or you tell libcurl that it is a post but would prefer to have libcurl instead get the data by using the regular read callback:

```
curl_easy_setopt(easy, CURLOPT_POST, 1L);
curl_easy_setopt(easy, CURLOPT_READFUNCTION, read_callback);
```

This "normal" POST will also set the request header `Content-Type: application/x-www-form-urlencoded`.

HTTP multipart formposts

A multipart formpost is still using the same HTTP method POST; the difference is only in the formatting of the request body. A multipart formpost is basically a series of separate "parts", separated by MIME-style boundary strings. There's no limit to how many parts you can send.

Each such part has a name, a set of headers and a few other properties.

libcurl offers a convenience function for constructing such a series of parts and to send that off to the server. `curl_formadd` is the function to build a formpost. Invoke it once for each part, and pass in arguments to it detailing the specifics and characteristics of that part. When

all parts you want to send have been added, you pass in the handle `curl_formadd` returned like this:

```
curl_easy_setopt(easy, CURLOPT_HTTPPOST, formposthandle);
```

HTTP PUT

TBD

Bindings

Creative people have written bindings or interfaces for various environments and programming languages. Using one of these allows you to take advantage of curl powers from within your favorite language or system. This is a list of all known interfaces as of this writing.

The bindings listed below are not part of the curl/libcurl distribution archives, but must be downloaded and installed separately.

Language	Site	Author(s)
Script Basic	http://scriptbasic.com/	Peter Verhas
C++	http://curlpp.org/	Jean-Philippe, Barrette-LaPierre
Ch/C++	https://chcurl.sourceforge.io/	Stephen Nestinger Jonathan Rogado
Cocoa (BBHTTP)	https://github.com/brunodecarvalho/BBHTTP	Bruno de Carvalho
Cocoa (CURLHandle)	https://github.com/karelia/curlhandle/	Dan Woo
D	https://dlang.org/library/std/net/curl.html	Kenneth Bogert
Delphi	https://github.com/Mercury13/curl4delphi	Mikhail Merkurye
Dylan	https://opendylan.org/	Chris Double
Eiffel	https://room.eiffel.com/library/curl	Eiffel Software
Falcon	http://www.falconpl.org/	Falcon
Gambas	https://gambas.sourceforge.io/	Gambas
glib/GTK+	http://atterer.org/glibcurl	Richard Atterer
Go	https://github.com/andelf/go-curl	ShuYu Wang

Guile	http://www.lonelycactus.com/guile-curl.html	Michael L Gran
Harbour	https://github.com/vszakats/harbour-core/tree/master/contrib/hbcurl	Viktor Szakáts
Haskell	https://hackage.haskell.org/package/curl	Galois, In
Java	https://github.com/pjlegato/curl-java	Paul Legato
Julia	https://github.com/JuliaWeb/LibCURL.jl	JuliaWeb
Lisp	https://common-lisp.net/project/cl-curl/	Liam Hea
Lua (luacurl)	http://luacurl.luaforge.net/	Alexander Marinov
Lua-cURL	https://github.com/Lua-cURL/Lua-cURLv3	Jürgen Hötzel, Alexey Melnichuk
.NET	https://github.com/masroore/CurlSharp	Masroor Ehsan Choudhur Jeffrey Phillips
NodeJS	https://github.com/JCMais/node-libcurl	Jonathan Cardoso Machado
OCaml	https://sourceforge.net/projects/ocurl/	Lars Nilsson
Pascal/Delphi/Kylix	https://curlpas.sourceforge.io/curlpas/	Jeffrey Pohlmeye
Perl	https://github.com/szbalint/WWW--Curl	Cris Bailiff and Bálint Szilakszi
PHP	https://php.net/curl	Sterling Hughes
PostgreSQL	https://github.com/pramsey/pgsql-http	Paul Ramsey
Python (PycURL)	https://github.com/pycurl/pycurl	Kjetil Jacobsen
R	https://cran.r-project.org/package=curl	Jeroen Ooms, Hadley Wickham, RStudio

Rexx	https://rexxcurl.sourceforge.io/	Mark Hessling
Ring	https://ring-lang.sourceforge.io/doc1.3/libcurl.html	Mahmoud Fayed
Ruby (curb)	https://github.com/taf2/curb	Ross Bamford
Ruby (ruby-curl-multi)	http://curl-multi.rubyforge.org/	Kristjan Petursson and Keith Rarick
Rust (curl-rust)	https://github.com/carllerche/curl-rust	Carl Lerche
Scheme Bigloo	https://www.metapaper.net/lisovsky/web/curl/	Kirill Lisovsky
Scilab	https://help.scilab.org/docs/current/fr_FR/getURL.html	Sylvestre Ledru
S-Lang	https://www.jedsoft.org/slang/modules/curl.html	John E Davis
Smalltalk	http://www.squeaksource.com/CurlPlugin/	Danil Osipchuk
SP-Forth	http://www.forth.org.ru/~ac/lib/lin/curl/	ygrek
Tcl	http://mirror.yellow5.com/tclcurl/	Andrés García
Visual Basic	https://sourceforge.net/projects/libcurl-vb/	Jeffrey Phillips
wxWidgets	https://wxcode.sourceforge.io/components/wxcurl/	Casey O'Donnell
Xojo	https://github.com/charonn0/RB-libcURL	Andrew Lambert

libcurl internals

libcurl is never finished and is not just an off-the-shelf product. It is very much a living project that is improved and modified on almost a daily basis. We depend on skilled and interested hackers to fix bugs and to add features.

This chapter is meant to describe internal details to aid keen libcurl hackers to learn some basic concepts on how libcurl works internally and thus possibly where to look for problems or where to add things when you want to make the library do something new.

Everything is multi

TBD

Different protocols "hooked in"

TBD

Everything is state machines

TBD

Name resolving

TBD

vtls

TBD

Index

.

/

<

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Y

Z